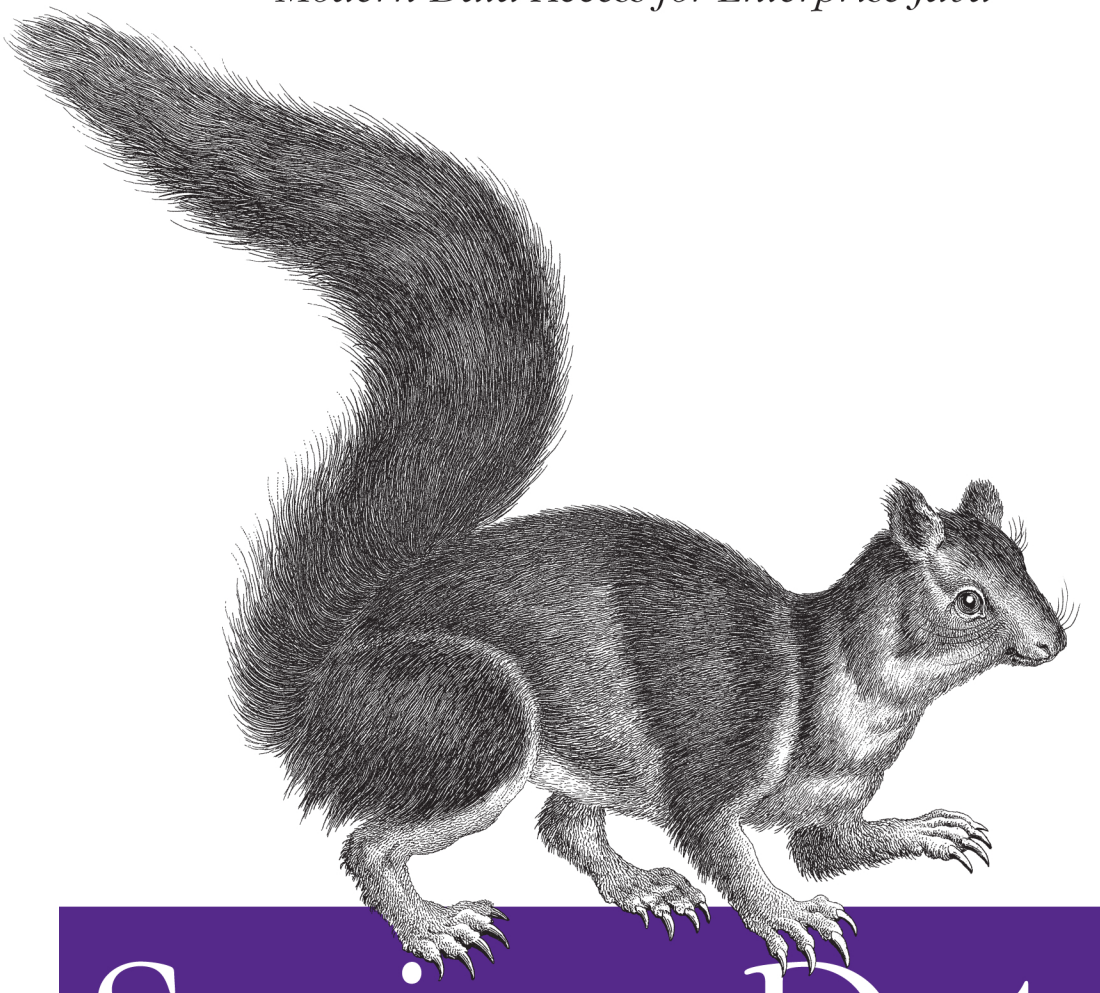
Modern Data Access for Enterprise Java



Spring Data

Mark Pollack, Oliver Gierke,
Thomas Risberg, Jonathan L. Brisbin,
& Michael Hunger

O'REILLY®

Spring Data

Modern Data Access for Enterprise Java

Mark Pollack, Oliver Gierke, Thomas Risberg,
Jon Brisbin, and Michael Hunger

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Spring Data
by Mark Pollack, Oliver Gierke, Thomas Risberg, Jon Brisbin, and Michael Hunger

Copyright © 2013 Mark Pollack, Oliver Gierke, Thomas Risberg, Jonathan L. Brisbin, Michael Hunger.
All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Meghan Blanchette
Production Editor: Kristen Borg
Proofreader: Rachel Monaghan

Indexer: Lucie Haskins
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

October 2012: First Edition.

Revision History for the First Edition:
2012-10-11 First release
See *http://oreilly.com/catalog/errata.csp?isbn=9781449323950* for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Spring Data*, the image of a giant squirrel, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-32395-0

[LSI]

1349968177

Thanks to my wife, Daniela, and sons, Gabriel and Alexandre, whose patience with me stealing time away for "the book" made it possible.

—Mark Pollack

I'd like to thank my family, friends, fellow musicians, and everyone I've had the pleasure to work with so far; the entire Spring Data and SpringSource team for this awesome journey; and last, but actually first of all, Sabine, for her inexhaustible love and support.

—Oliver Gierke

To my wife, Carol, and my son, Alex, thank you for enriching my life and for all your support and encouragement.

—Thomas Risberg

To my wife, Tisha; my sons, Jack, Ben, and Daniel; and my daughters, Morgan and Hannah. Thank you for your love, support, and patience. All this wouldn't be worth it without you.

—Jon Brisbin

My special thanks go to Rod and Emil for starting the Spring Data project and to Oliver for making it great. My family is always very supportive of my crazy work; I'm very grateful to have such understanding women around me.

—Michael Hunger

I'd like to thank my wife, Nanette, and my kids for their support, patience, and understanding. Thanks also to Rod and my colleagues on the Spring Data team for making all of this possible.

—David Turanski

Table of Contents

Foreword ..... **xiii**

Preface ..... **xv**

Part I. Background

1. The Spring Data Project **3**
NoSQL Data Access for Spring Developers 3
General Themes 5
The Domain 6
The Sample Code 6
Importing the Source Code into Your IDE 7

2. Repositories: Convenient Data Access Layers **13**
Quick Start 13
Defining Query Methods 16
Query Lookup Strategies 16
Query Derivation 17
Pagination and Sorting 18
Defining Repositories 19
Fine-Tuning Repository Interfaces 20
Manually Implementing Repository Methods 21
IDE Integration 22
IntelliJ IDEA 24

3. Type-Safe Querying Using Querydsl **27**
Introduction to Querydsl 27
Generating the Query Metamodel 30
Build System Integration 30
Supported Annotation Processors 31

Querying Stores Using Querydsl 32
Integration with Spring Data Repositories 32
Executing Predicates 33
Manually Implementing Repositories 34

Part II. Relational Databases

4. JPA Repositories 37
The Sample Project 37
The Traditional Approach 42
Bootstrapping the Sample Code 44
Using Spring Data Repositories 47
Transactionality 50
Repository Querydsl Integration 51

5. Type-Safe JDBC Programming with Querydsl SQL 53
The Sample Project and Setup 53
The HyperSQL Database 54
The SQL Module of Querydsl 54
Build System Integration 58
The Database Schema 59
The Domain Implementation of the Sample Project 60
The QueryDslJdbcTemplate 63
Executing Queries 64
The Beginning of the Repository Implementation 64
Querying for a Single Object 65
The OneToManyResultSetExtractor Abstract Class 67
The CustomerListExtractor Implementation 68
The Implementations for the RowMappers 69
Querying for a List of Objects 71
Insert, Update, and Delete Operations 71
Inserting with the SQLInsertClause 71
Updating with the SQLUpdateClause 72
Deleting Rows with the SQLDeleteClause 73

Part III. NoSQL

6. MongoDB: A Document Store 77
MongoDB in a Nutshell 77
Setting Up MongoDB 78
Using the MongoDB Shell 79

The MongoDB Java Driver 80
Setting Up the Infrastructure Using the Spring Namespace 81
The Mapping Subsystem 83
The Domain Model 83
Setting Up the Mapping Infrastructure 89
Indexing 91
Customizing Conversion 91
MongoTemplate 94
Mongo Repositories 96
Infrastructure Setup 96
Repositories in Detail 97
Mongo Querydsl Integration 99

7. Neo4j: A Graph Database 101
Graph Databases 101
Neo4j 102
Spring Data Neo4j Overview 105
Modeling the Domain as a Graph 106
Persisting Domain Objects with Spring Data Neo4j 111
Neo4jTemplate 112
Combining Graph and Repository Power 113
Basic Graph Repository Operations 115
Derived and Annotated Finder Methods 116
Advanced Graph Use Cases in the Example Domain 119
Multiple Roles for a Single Node 119
Product Categories and Tags as Examples for In-Graph Indexes 120
Leverage Similar Interests (Collaborative Filtering) 121
Recommendations 121
Transactions, Entity Life Cycle, and Fetch Strategies 122
Advanced Mapping Mode 123
Working with Neo4j Server 124
Continuing From Here 125

8. Redis: A Key/Value Store 127
Redis in a Nutshell 127
Setting Up Redis 127
Using the Redis Shell 128
Connecting to Redis 129
Object Conversion 130
Object Mapping 132
Atomic Counters 134
Pub/Sub Functionality 135
Listening and Responding to Messages 135

Using Spring's Cache Abstraction with Redis 136

Part IV. Rapid Application Development

9. Persistence Layers with Spring Roo 141
A Brief Introduction to Roo 141
Roo's Persistence Layers 143
Quick Start 143
Using Roo from the Command Line 143
Using Roo with Spring Tool Suite 145
A Spring Roo JPA Repository Example 147
Creating the Project 147
Setting Up JPA Persistence 148
Creating the Entities 148
Defining the Repositories 150
Creating the Web Layer 150
Running the Example 151
A Spring Roo MongoDB Repository Example 152
Creating the Project 153
Setting Up MongoDB Persistence 153
Creating the Entities 153
Defining the Repositories 154
Creating the Web Layer 154
Running the Example 154

10. REST Repository Exporter 157
The Sample Project 158
Interacting with the REST Exporter 160
Accessing Products 162
Accessing Customers 165
Accessing Orders 169

Part V. Big Data

11. Spring for Apache Hadoop 175
Challenges Developing with Hadoop 176
Hello World 177
Hello World Revealed 179
Hello World Using Spring for Apache Hadoop 183
Scripting HDFS on the JVM 187
Combining HDFS Scripting and Job Submission 190

Job Scheduling 191
Scheduling MapReduce Jobs with a TaskScheduler 191
Scheduling MapReduce Jobs with Quartz 192

12. Analyzing Data with Hadoop 195
Using Hive 195
Hello World 196
Running a Hive Server 197
Using the Hive Thrift Client 198
Using the Hive JDBC Client 201
Apache Logfile Analysis Using Hive 202
Using Pig 204
Hello World 205
Running a PigServer 207
Controlling Runtime Script Execution 209
Calling Pig Scripts Inside Spring Integration Data Pipelines 211
Apache Logfile Analysis Using Pig 212
Using HBase 214
Hello World 214
Using the HBase Java Client 215

13. Creating Big Data Pipelines with Spring Batch and Spring Integration 219
Collecting and Loading Data into HDFS 219
An Introduction to Spring Integration 220
Copying Logfiles 222
Event Streams 226
Event Forwarding 229
Management 230
An Introduction to Spring Batch 232
Processing and Loading Data from a Database 234
Hadoop Workflows 238
Spring Batch Support for Hadoop 238
Wordcount as a Spring Batch Application 240
Hive and Pig Steps 242
Exporting Data from HDFS 243
From HDFS to JDBC 243
From HDFS to MongoDB 249
Collecting and Loading Data into Splunk 250

Part VI. Data Grids

14. GemFire: A Distributed Data Grid 255
GemFire in a Nutshell 255
Caches and Regions 257
How to Get GemFire 257
Configuring GemFire with the Spring XML Namespace 258
Cache Configuration 258
Region Configuration 263
Cache Client Configuration 265
Cache Server Configuration 267
WAN Configuration 267
Disk Store Configuration 268
Data Access with GemfireTemplate 269
Repository Usage 271
POJO Mapping 271
Creating a Repository 272
PDX Serialization 272
Continuous Query Support 273

Bibliography 275

Index 277

Foreword

We live in interesting times. New business processes are driving new requirements. Familiar assumptions are under threat—among them, that the relational database should be the default choice for persistence. While this is now widely accepted, it is far from clear how to proceed effectively into the new world.

A proliferation of data store choices creates fragmentation. Many newer stores require more developer effort than Java developers are used to regarding data access, pushing into the application things customarily done in a relational database.

This book helps you make sense of this new reality. It provides an excellent overview of today's storage world in the context of today's hardware, and explains why NoSQL stores are important in solving modern business problems.

Because of the language's identification with the often-conservative enterprise market (and perhaps also because of the sophistication of Java object-relational mapping [ORM] solutions), Java developers have traditionally been poorly served in the NoSQL space. Fortunately, this is changing, making this an important and timely book. Spring Data is an important project, with the potential to help developers overcome new challenges.

Many of the values that have made Spring the preferred platform for enterprise Java developers deliver particular benefit in a world of fragmented persistence solutions. Part of the value of Spring is how it brings consistency (without descending to a lowest common denominator) in its approach to different technologies with which it integrates. A distinct "Spring way" helps shorten the learning curve for developers and simplifies code maintenance. If you are already familiar with Spring, you will find that Spring Data eases your exploration and adoption of unfamiliar stores. If you aren't already familiar with Spring, this is a good opportunity to see how Spring can simplify your code and make it more consistent.

The authors are uniquely qualified to explain Spring Data, being the project leaders. They bring a mix of deep Spring knowledge and involvement and intimate experience with a range of modern data stores. They do a good job of explaining the motivation of Spring Data and how it continues the mission Spring has long pursued regarding data access. There is valuable coverage of how Spring Data works with other parts of

Spring, such as Spring Integration and Spring Batch. The book also provides much value that goes beyond Spring—for example, the discussions of the repository concept, the merits of type-safe querying, and why the Java Persistence API (JPA) is not appropriate as a general data access solution.

While this is a book about data access rather than working with NoSQL, many of you will find the NoSQL material most valuable, as it introduces topics and code with which you are likely to be less familiar. All content is up to the minute, and important topics include document databases, graph databases, key/value stores, Hadoop, and the Gemfire data fabric.

We programmers are practical creatures and learn best when we can be hands-on. The book has a welcome practical bent. Early on, the authors show how to get the sample code working in the two leading Java integrated development environments (IDEs), including handy screenshots. They explain requirements around database drivers and basic database setup. I applaud their choice of hosting the sample code on GitHub, making it universally accessible and browsable. Given the many topics the book covers, the well-designed examples help greatly to tie things together.

The emphasis on practical development is also evident in the chapter on Spring Roo, the rapid application development (RAD) solution from the Spring team. Most Roo users are familiar with how Roo can be used with a traditional JPA architecture; the authors show how Roo's productivity can be extended beyond relational databases.

When you've finished this book, you will have a deeper understanding of why modern data access is becoming more specialized and fragmented, the major categories of NoSQL data stores, how Spring Data can help Java developers operate effectively in this new environment, and where to look for deeper information on individual topics in which you are particularly interested. Most important, you'll have a great start to your own exploration in code!

—Rod Johnson
Creator, Spring Framework

Preface

Overview of the New Data Access Landscape

The data access landscape over the past seven or so years has changed dramatically. Relational databases, the heart of storing and processing data in the enterprise for over 30 years, are no longer the only game in town. The past seven years have seen the birth —and in some cases the death—of many alternative data stores that are being used in mission-critical enterprise applications. These new data stores have been designed specifically to solve data access problems that relational database can't handle as effectively.

An example of a problem that pushes traditional relational databases to the breaking point is scale. How do you store hundreds or thousands of terabytes (TB) in a relational database? The answer reminds us of the old joke where the patient says, "Doctor, it hurts when I do this," and the doctor says, "Then don't do that!" Jokes aside, what is driving the need to store this much data? In 2001, IDC reported that "the amount of information created and replicated will surpass 1.8 zettabytes and more than double every two years."[1] New data types range from media files to logfiles to sensor data (RFID, GPS, telemetry...) to tweets on Twitter and posts on Facebook. While data that is stored in relational databases is still crucial to the enterprise, these new types of data are not being stored in relational databases.

While general consumer demands drive the need to store large amounts of media files, enterprises are finding it important to store and analyze many of these new sources of data. In the United States, companies in all sectors have at least 100 TBs of stored data and many have more than 1 petabyte (PB).[2] The general consensus is that there are significant bottom-line benefits for businesses to continually analyze this data. For example, companies can better understand the behavior of their products if the products themselves are sending "phone home" messages about their health. To better understand their customers, companies can incorporate social media data into their decision-making processes. This has led to some interesting mainstream media

1. IDC; *Extracting Value from Chaos*. 2011.
2. IDC; US Bureau of Labor Statistics

reports—for example, on why Orbitz shows more expensive hotel options to Mac users and how Target can predict when one of its customers will soon give birth, allowing the company to mail coupon books to the customer's home before public birth records are available.

Big data generally refers to the process in which large quantities of data are stored, kept in raw form, and continually analyzed and combined with other data sources to provide a deeper understanding of a particular domain, be it commercial or scientific in nature.

Many companies and scientific laboratories had been performing this process before the term *big data* came into fashion. What makes the current process different from before is that the value derived from the intelligence of data analytics is higher than the hardware costs. It is no longer necessary to buy a 40K per CPU box to perform this type of data analysis; clusters of commodity hardware now cost $1k per CPU. For large datasets, the cost of storage area network (SAN) or network area storage (NAS) becomes prohibitive: $1 to $10 per gigabyte, while local disk costs only $0.05 per gigabyte with replication built into the database instead of the hardware. Aggregate data transfer rates for clusters of commodity hardware that use local disk are also significantly higher than SAN- or NAS-based systems—500 times faster for similarly priced systems. On the software side, the majority of the new data access technologies are open source. While open source does not mean zero cost, it certainly lowers the barrier for entry and overall cost of ownership versus the traditional commercial software offerings in this space.

Another problem area that new data stores have identified with relational databases is the relational data model. If you are interested in analyzing the social graph of millions of people, doesn't it sound quite natural to consider using a graph database so that the implementation more closely models the domain? What if requirements are continually driving you to change your relational database management system (RDBMS) schema and object-relational mapping (ORM) layer? Perhaps a "schema-less" document database will reduce the object mapping complexity and provide a more easily evolvable system as compared to the more rigid relational model. While each of the new databases is unique in its own way, you can provide a rough taxonomy across most of them based on their data models. The basic camps they fall into are:

Key/value
: A familiar data model, much like a hashtable.

Column family
: An extended key/value data model in which the value data type can also be a sequence of key/value pairs.

Document
: Collections that contain semistructured data, such as XML or JSON.

Graph
: Based on graph theory. The data model has nodes and edges, each of which may have properties.

The general name under which these new databases have become grouped is "NoSQL databases." In retrospect, this name, while catchy, isn't very accurate because it seems to imply that you can't query the database, which isn't true. It reflects the basic shift away from the relational data model as well as a general shift away from ACID (atomicity, consistency, isolation, durability) characteristics of relational databases.

One of the driving factors for the shift away from ACID characteristics is the emergence of applications that place a higher priority on scaling writes and having a partially functioning system even when parts of the system have failed. While scaling reads in a relational database can be achieved through the use of in-memory caches that front the database, scaling writes is much harder. To put a label on it, these new applications favor a system that has so-called "BASE" semantics, where the acronym represents *basically available, scalable, eventually consistent*. Distributed data grids with a key/value data model generally have not been grouped into this new wave of NoSQL databases. However, they offer similar features to NoSQL databases in terms of the scale of data they can handle as well as distributed computation features that colocate computing power and data.

As you can see from this brief introduction to the new data access landscape, there is a revolution taking place, which for data geeks is quite exciting. Relational databases are not dead; they are still central to the operation of many enterprises and will remain so for quite some time. The trends, though, are very clear: new data access technologies are solving problems that traditional relational databases can't, so we need to broaden our skill set as developers and have a foot in both camps.

The Spring Framework has a long history of simplifying the development of Java applications, in particular for writing RDBMS-based data access layers that use Java database connectivity (JDBC) or object-relational mappers. In this book we aim to help developers get a handle on how to effectively develop Java applications across a wide range of these new technologies. The Spring Data project directly addresses these new technologies so that you can extend your existing knowledge of Spring to them, or perhaps learn more about Spring as a byproduct of using Spring Data. However, it doesn't leave the relational database behind. Spring Data also provides an extensive set of new features to Spring's RDBMS support.

How to Read This Book

This book is intended to give you a hands-on introduction to the Spring Data project, whose core mission is to enable Java developers to use state-of-the-art data processing and manipulation tools but also use traditional databases in a state-of-the-art manner. We'll start by introducing you to the project, outlining the primary motivation of SpringSource and the team. We'll also describe the domain model of the sample projects that accommodate each of the later chapters, as well as how to access and set up the code (Chapter 1).

We'll then discuss the general concepts of Spring Data repositories, as they are a common theme across the various store-specific parts of the project (Chapter 2). The same applies to Querydsl, which is discussed in general in Chapter 3. These two chapters provide a solid foundation to explore the store specific integration of the repository abstraction and advanced query functionality.

To start Java developers in well-known terrain, we'll then spend some time on traditional persistence technologies like JPA (Chapter 4) and JDBC (Chapter 5). Those chapters outline what features the Spring Data modules add on top of the already existing JPA and JDBC support provided by Spring.

After we've finished that, we introduce some of the NoSQL stores supported by the Spring Data project: MongoDB as an example of a document database (Chapter 6), Neo4j as an example of a graph database (Chapter 7), and Redis as an example of a key/value store (Chapter 8). HBase, a column family database, is covered in a later chapter (Chapter 12). These chapters outline mapping domain classes onto the store-specific data structures, interacting easily with the store through the provided application programming interface (API), and using the repository abstraction.

We'll then introduce you to the Spring Data REST exporter (Chapter 10) as well as the Spring Roo integration (Chapter 9). Both projects build on the repository abstraction and allow you to easily export Spring Data–managed entities to the Web, either as a representational state transfer (REST) web service or as backing to a Spring Roo–built web application.

The book next takes a tour into the world of big data—Hadoop and Spring for Apache Hadoop in particular. It will introduce you to using cases implemented with Hadoop and show how the Spring Data module eases working with Hadoop significantly (Chapter 11). This leads into a more complex example of building a big data pipeline using Spring Batch and Spring Integration—projects that come nicely into play in big data processing scenarios (Chapter 12 and Chapter 13).

The final chapter discusses the Spring Data support for Gemfire, a distributed data grid solution (Chapter 14).

Conventions Used in This Book

The following typographical conventions are used in this book:

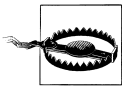
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Spring Data* by Mark Pollack, Oliver Gierke, Thomas Risberg, Jon Brisbin, and Michael Hunger (O'Reilly). Copyright 2013 Mark Pollack, Oliver Gierke, Thomas Risberg, Jonathan L. Brisbin, and Michael Hunger, 978-1-449-32395-0."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

The code samples are posted on GitHub.

Safari® Books Online

Safari Books Online (*www.safaribooksonline.com*) is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of product mixes and pricing programs for organizations, government agencies, and individuals. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://oreil.ly/spring-data-1e*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

We would like to thank Rod Johnson and Emil Eifrem for starting what was to become the Spring Data project.

A big thank you goes to David Turanski for pitching in and helping out with the GemFire chapter. Thank you to Richard McDougall for the big data statistics used in the introduction, and to Costin Leau for help with writing the Hadoop sample applications.

We would also like to thank O'Reilly Media, especially Meghan Blanchette for guiding us through the project, production editor Kristen Borg, and copyeditor Rachel Monaghan. Thanks to Greg Turnquist, Joris Kuipers, Johannes Hiemer, Joachim Arrasz, Stephan Hochdörfer, Mark Spritzler, Jim Webber, Lasse Westh-Nielsen, and all other technical reviewers for their feedback. Thank you to the community around the project for sending feedback and issues so that we could constantly improve. Last but not least, thanks to our friends and families for their patience, understanding, and support.

PART I

Background

CHAPTER 1

The Spring Data Project

The Spring Data project was coined at Spring One 2010 and originated from a hacking session of Rod Johnson (SpringSource) and Emil Eifrem (Neo Technologies) early that year. They were trying to integrate the Neo4j graph database with the Spring Framework and evaluated different approaches. The session created the foundation for what would eventually become the very first version of the Neo4j module of Spring Data, a new SpringSource project aimed at supporting the growing interest in NoSQL data stores, a trend that continues to this day.

Spring has provided sophisticated support for traditional data access technologies from day one. It significantly simplified the implementation of data access layers, regardless of whether JDBC, Hibernate, TopLink, JDO, or iBatis was used as persistence technology. This support mainly consisted of simplified infrastructure setup and resource management as well as exception translation into Spring's `DataAccessException`s. This support has matured over the years and the latest Spring versions contained decent upgrades to this layer of support.

The traditional data access support in Spring has targeted relational databases only, as they were the predominant tool of choice when it came to data persistence. As NoSQL stores enter the stage to provide reasonable alternatives in the toolbox, there's room to fill in terms of developer support. Beyond that, there are yet more opportunities for improvement even for the traditional relational stores. These two observations are the main drivers for the Spring Data project, which consists of dedicated modules for NoSQL stores as well as JPA and JDBC modules with additional support for relational databases.

NoSQL Data Access for Spring Developers

Although the term NoSQL is used to refer to a set of quite young data stores, all of the stores have very different characteristics and use cases. Ironically, it's the nonfeature (the lack of support for running queries using SQL) that actually named this group of databases. As these stores have quite different traits, their Java drivers have completely

different APIs to leverage the stores' special traits and features. Trying to abstract away their differences would actually remove the benefits each NoSQL data store offers. A graph database should be chosen to store highly interconnected data. A document database should be used for tree and aggregate-like data structures. A key/value store should be chosen if you need cache-like functionality and access patterns.

With the JPA, the Java EE (Enterprise Edition) space offers a persistence API that could have been a candidate to front implementations of NoSQL databases. Unfortunately, the first two sentences of the specification already indicate that this is probably not working out:

> This document is the specification of the Java API for the management of persistence and object/relational mapping with Java EE and Java SE. The technical objective of this work is to provide an object/relational mapping facility for the Java application developer using a Java domain model to manage a relational database.

This theme is clearly reflected in the specification later on. It defines concepts and APIs that are deeply connected to the world of relational persistence. An `@Table` annotation would not make a lot of sense for NoSQL databases, nor would `@Column` or `@JoinColumn`. How should one implement the transaction API for stores like MongoDB, which essentially do not provide transactional semantics spread across multidocument manipulations? So implementing a JPA layer on top of a NoSQL store would result in a profile of the API at best.

On the other hand, all the special features NoSQL stores provide (geospatial functionality, map-reduce operations, graph traversals) would have to be implemented in a proprietary fashion anyway, as JPA simply does not provide abstractions for them. So we would essentially end up in a worst-of-both-worlds scenario—the parts that can be implemented behind JPA plus additional proprietary features to reenable store-specific features.

This context rules out JPA as a potential abstraction API for these stores. Still, we would like to see the programmer productivity and programming model consistency known from various Spring ecosystem projects to simplify working with NoSQL stores. This led the Spring Data team to declare the following mission statement:

> Spring Data provides a familiar and consistent Spring-based programming model for NoSQL and relational stores while retaining store-specific features and capabilities.

So we decided to take a slightly different approach. Instead of trying to abstract all stores behind a single API, the Spring Data project provides a consistent programming model across the different store implementations using patterns and abstractions already known from within the Spring Framework. This allows for a consistent experience when you're working with different stores.

General Themes

A core theme of the Spring Data project available for all of the stores is support for configuring resources to access the stores. This support is mainly implemented as XML namespace and support classes for Spring JavaConfig and allows us to easily set up access to a Mongo database, an embedded Neo4j instance, and the like. Also, integration with core Spring functionality like JMX is provided, meaning that some stores will expose statistics through their native API, which will be exposed to JMX via Spring Data.

Most of the NoSQL Java APIs do not provide support to map domain objects onto the stores' data abstractions (documents in MongoDB; nodes and relationships for Neo4j). So, when working with the native Java drivers, you would usually have to write a significant amount of code to map data onto the domain objects of your application when reading, and vice versa on writing. Thus, a very core part of the Spring Data modules is a mapping and conversion API that allows obtaining metadata about domain classes to be persistent and enables the actual conversion of arbitrary domain objects into store-specific data types.

On top of that, we'll find opinionated APIs in the form of template pattern implementations already well known from Spring's `JdbcTemplate`, `JmsTemplate`, etc. Thus, there is a `RedisTemplate`, a `MongoTemplate`, and so on. As you probably already know, these templates offer helper methods that allow us to execute commonly needed operations like persisting an object with a single statement while automatically taking care of appropriate resource management and exception translation. Beyond that, they expose callback APIs that allow you to access the store-native APIs while still getting exceptions translated and resources managed properly.

These features already provide us with a toolbox to implement a data access layer like we're used to with traditional databases. The upcoming chapters will guide you through this functionality. To ease that process even more, Spring Data provides a repository abstraction on top of the template implementation that will reduce the effort to implement data access objects to a plain interface definition for the most common scenarios like performing standard CRUD operations as well as executing queries in case the store supports that. This abstraction is actually the topmost layer and blends the APIs of the different stores as much as reasonably possible. Thus, the store-specific implementations of it share quite a lot of commonalities. This is why you'll find a dedicated chapter (Chapter 2) introducing you to the basic programming model.

Now let's take a look at our sample code and the domain model that we will use to demonstrate the features of the particular store modules.

The Domain

To illustrate how to work with the various Spring Data modules, we will be using a sample domain from the ecommerce sector (see Figure 1-1). As NoSQL data stores usually have a dedicated sweet spot of functionality and applicability, the individual chapters might tweak the actual implementation of the domain or even only partially implement it. This is not to suggest that you have to model the domain in a certain way, but rather to emphasize which store might actually work better for a given application scenario.

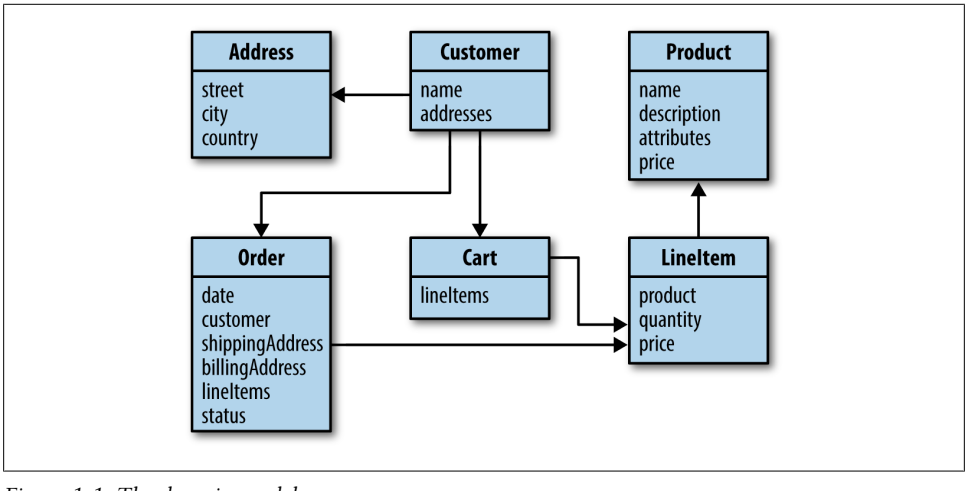


Figure 1-1. The domain model

At the core of our model, we have a customer who has basic data like a first name, a last name, an email address, and a set of addresses in turn containing street, city, and country. We also have products that consist of a name, a description, a price, and arbitrary attributes. These abstractions form the basis of a rudimentary CRM (customer relationship management) and inventory system. On top of that, we have orders a customer can place. An order contains the customer who placed it, shipping and billing addresses, the date the order was placed, an order status, and a set of line items. These line items in turn reference a particular product, the number of products to be ordered, and the price of the product.

The Sample Code

The sample code for this book can be found on GitHub. It is a Maven project containing a module per chapter. It requires either a Maven 3 installation on your machine or an IDE capable of importing Maven projects such as the Spring Tool Suite (STS). Getting the code is as simple as cloning the repository:

```
$ cd ~/dev
$ git clone https://github.com/SpringSource/spring-data-book.git
Cloning into 'spring-data-book'...
remote: Counting objects: 253, done.
remote: Compressing objects: 100% (137/137), done.
Receiving objects: 100% (253/253), 139.99 KiB | 199 KiB/s, done.
remote: Total 253 (delta 91), reused 219 (delta 57)
Resolving deltas: 100% (91/91), done.
$ cd spring-data-book
```

You can now build the code by executing Maven from the command line as follows:

```
$ mvn clean package
```

This will cause Maven to resolve dependencies, compile and test code, execute tests, and package the modules eventually.

Importing the Source Code into Your IDE

STS/Eclipse

STS ships with the m2eclipse plug-in to easily work with Maven projects right inside your IDE. So, if you have it already downloaded and installed (have a look at Chapter 3 for details), you can choose the Import option of the File menu. Select the Existing Maven Projects option from the dialog box, shown in Figure 1-2.

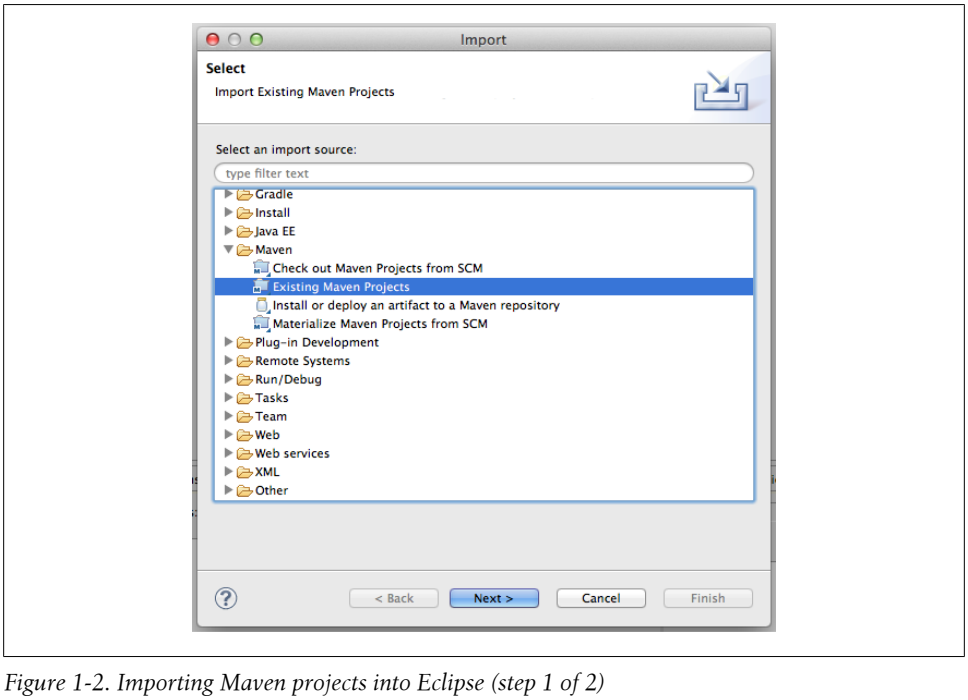


Figure 1-2. Importing Maven projects into Eclipse (step 1 of 2)

In the next window, select the folder in which you've just checked out the project using the Browse button. After you've done so, the pane right below should fill with the individual Maven modules listed and checked (Figure 1-3). Proceed by clicking on Finish, and STS will import the selected Maven modules into your workspace. It will also resolve the necessary dependencies and source folder according to the *pom.xml* file in the module's root directory.

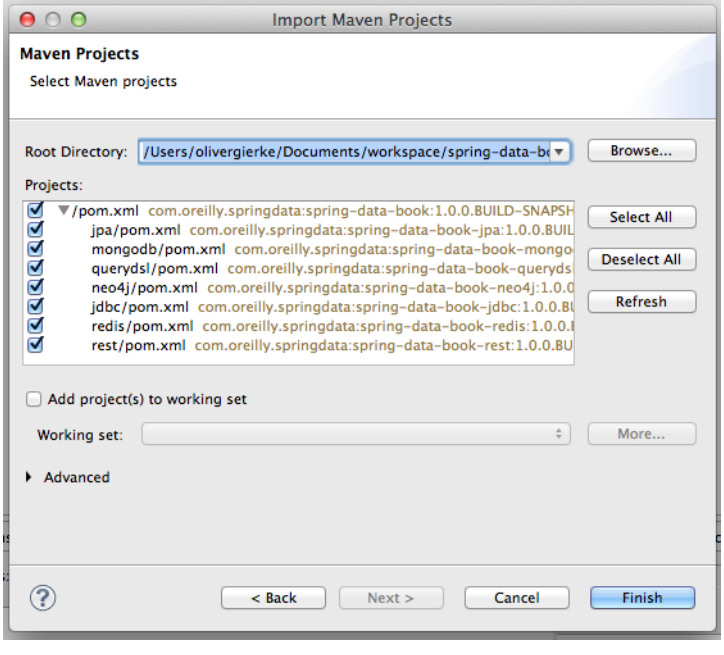


Figure 1-3. Importing Maven projects into Eclipse (step 2 of 2)

You should eventually end up with a Package or Project Explorer looking something like Figure 1-4. The projects should compile fine and contain no red error markers.

The projects using Querydsl (see Chapter 5 for details) might still carry a red error marker. This is due to the m2eclipse plug-in needing additional information about when to execute the Querydsl-related Maven plug-ins in the IDE build life cycle. The integration for that can be installed from the m2e-querydsl extension update site; you'll find the most recent version of it at the project home page. Copy the link to the latest version listed there (0.0.3, at the time of this writing) and add it to the list of available update sites, as shown in Figure 1-5. Installing the feature exposed through that update site, restarting Eclipse, and potentially updating the Maven project configuration (right-click on the project→Maven→Update Project) should let you end up with all the projects without Eclipse error markers and building just fine.

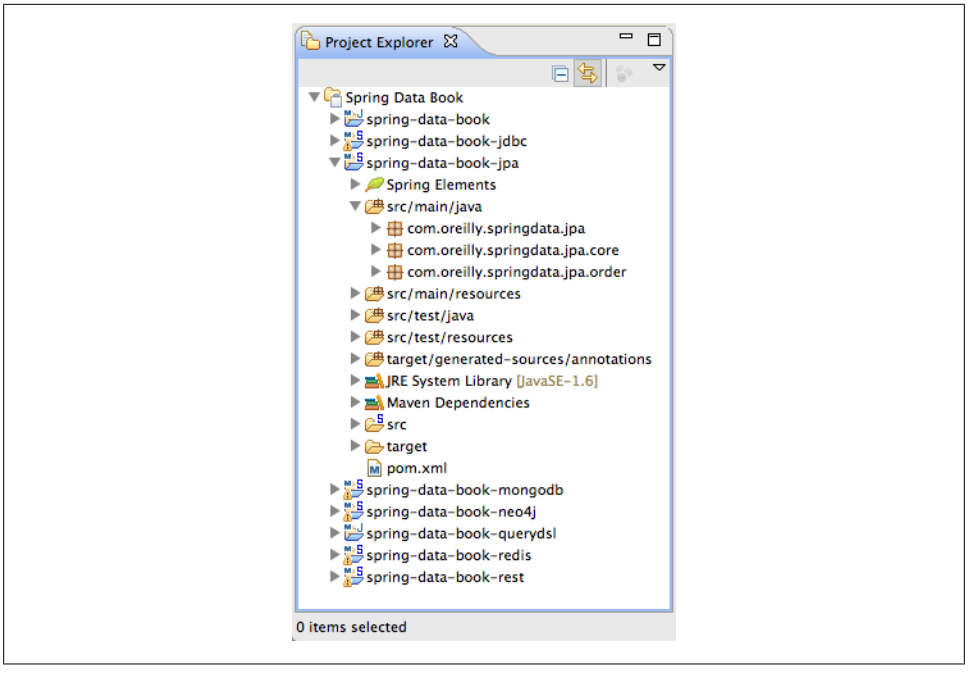


Figure 1-4. Eclipse Project Explorer with import finished

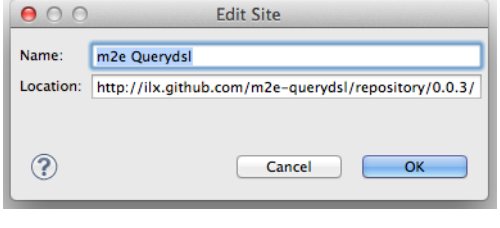


Figure 1-5. Adding the m2e-querydsl update site

IntelliJ IDEA

IDEA is able to open Maven project files directly without any further setup needed. Select the Open Project menu entry to show the dialog box (see Figure 1-6).

The IDE opens the project and fetches needed dependencies. In the next step (shown in Figure 1-7), it detects used frameworks (like the Spring Framework, JPA, WebApp); use the Configure link in the pop up or the Event Log to configure them.

The project is then ready to be used. You will see the Project view and the Maven Projects view, as shown in Figure 1-8. Compile the project as usual.

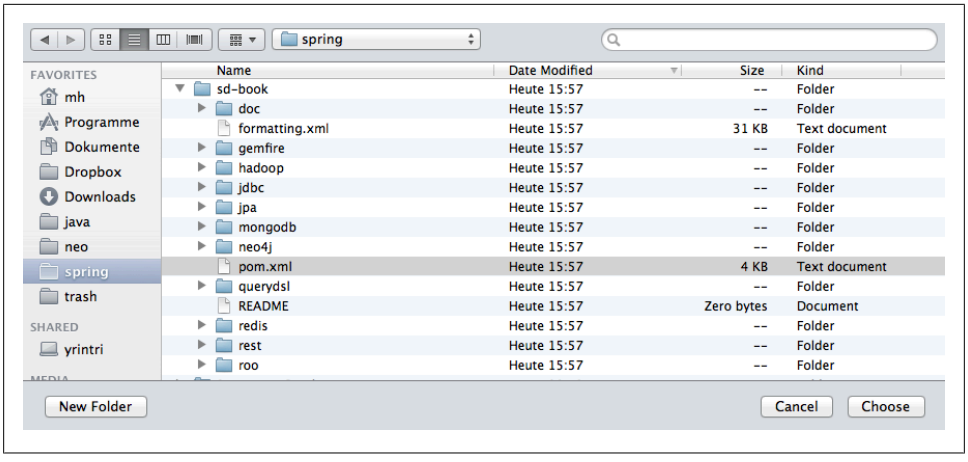


Figure 1-6. Importing Maven projects into IDEA (step 1 of 2)

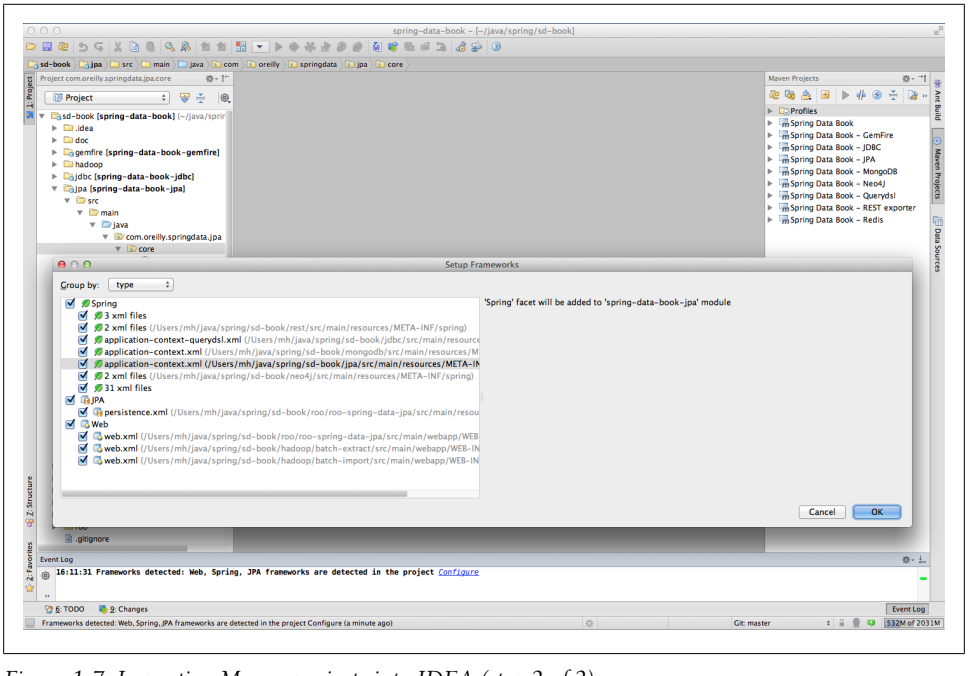


Figure 1-7. Importing Maven projects into IDEA (step 2 of 2)

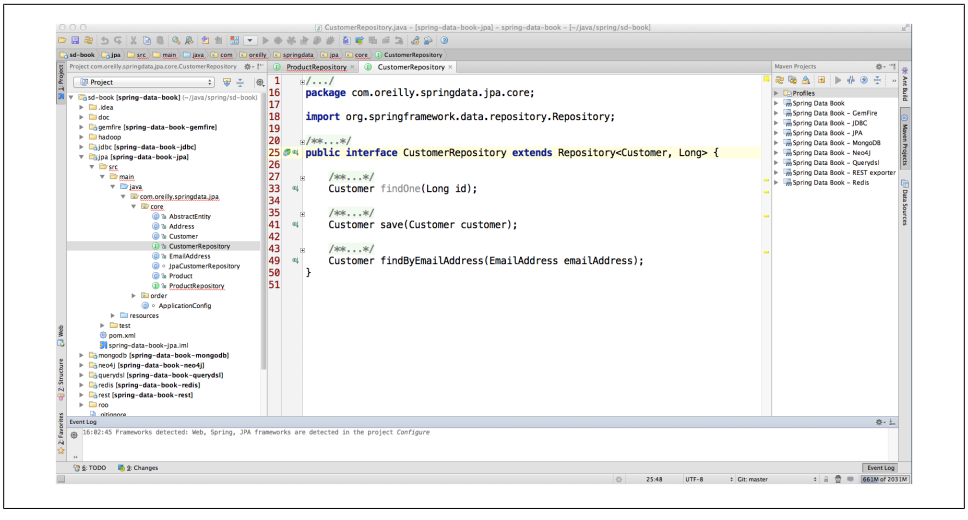


Figure 1-8. IDEA with the Spring Data Book project opened

Next you must add JPA support in the Spring Data JPA module to enable finder method completion and error checking of repositories. Just right-click on the module and choose Add Framework. In the resulting dialog box, check JavaEE Persistence support and select Hibernate as the persistence provider (Figure 1-9). This will create a *src/main/java/resources/META-INF/persistence.xml* file with just a persistence-unit setup.

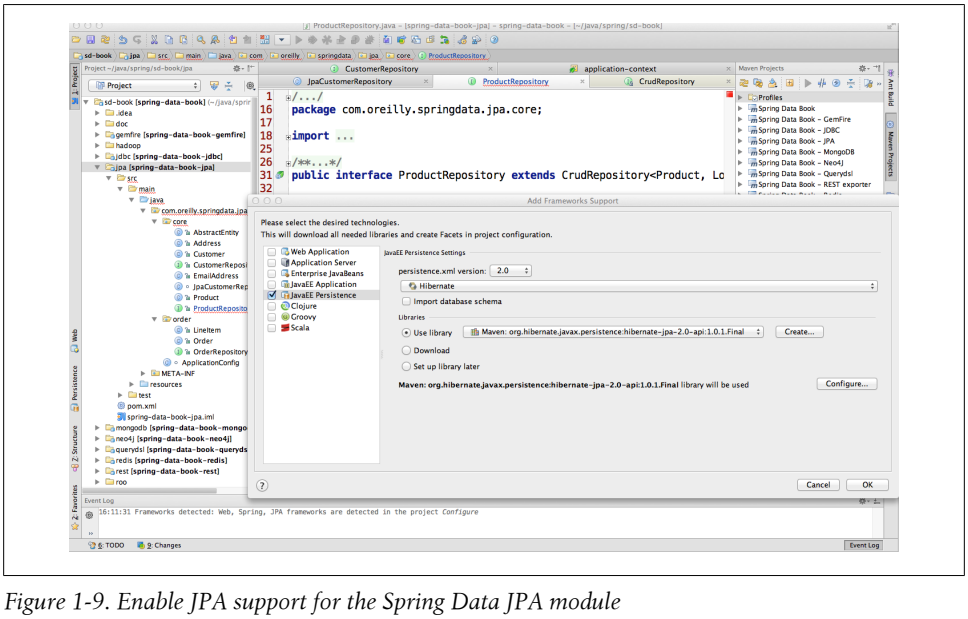


Figure 1-9. Enable JPA support for the Spring Data JPA module

CHAPTER 2

Repositories: Convenient Data Access Layers

Implementing the data access layer of an application has been cumbersome for quite a while. Too much boilerplate code had to be written. Domain classes were anemic and not designed in a real object-oriented or domain-driven manner. The goal of the repository abstraction of Spring Data is to reduce the effort required to implement data access layers for various persistence stores significantly. The following sections will introduce the core concepts and interfaces of Spring Data repositories. We will use the Spring Data JPA module as an example and discuss the basic concepts of the repository abstraction. For other stores, make sure you adapt the examples accordingly.

Quick Start

Let's take the `Customer` domain class from our domain that will be persisted to an arbitrary store. The class might look something like Example 2-1.

Example 2-1. The Customer domain class

```
public class Customer {

  private Long id;
  private String firstname;
  private String lastname;
  private EmailAddress emailAddress;
  private Address address;

  …
}
```

A traditional approach to a data access layer would now require you to at least implement a repository class that contains necessary CRUD (Create, Read, Update, and Delete) methods as well as query methods to access subsets of the entities stored by applying restrictions on them. The Spring Data repository approach allows you to get

rid of most of the implementation code and instead start with a plain interface definition for the entity's repository, as shown in Example 2-2.

Example 2-2. The CustomerRepository interface definition

```
public interface CustomerRepository extends Repository<Customer, Long> {
  …
}
```

As you can see, we extend the Spring Data `Repository` interface, which is just a generic marker interface. Its main responsibility is to allow the Spring Data infrastructure to pick up all user-defined Spring Data repositories. Beyond that, it captures the type of the domain class managed alongside the type of the ID of the entity, which will come in quite handy at a later stage. To trigger the autodiscovery of the interfaces declared, we use either the `<repositories />` element of the store-specific XML namespace (Example 2-3) or the related `@Enable…Repositories` annotation in case we're using JavaConfig (Example 2-4). In our sample case, we will use JPA. We just need to configure the XML element's `base-package` attribute with our root package so that Spring Data will scan it for repository interfaces. The annotation can also get a dedicated package configured to scan for interfaces. Without any further configuration given, it will simply inspect the package of the annotated class.

Example 2-3. Activating Spring Data repository support using XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jpa="http://www.springframework.org/schema/data/jpa"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/jpa
    http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

  <jpa:repositories base-package="com.acme.**.repository" />

</beans>
```

Example 2-4. Activating Spring Data repository support using Java Config

```
@Configuration
@EnableJpaRepositories
class ApplicationConfig {

}
```

Both the XML and JavaConfig configuration will need to be enriched with store-specific infrastructure bean declarations, such as a JPA `EntityManagerFactory`, a `DataSource`, and the like. For other stores, we simply use the corresponding namespace elements or annotations. The configuration snippet, shown in Example 2-5, will now cause the Spring Data repositories to be found, and Spring beans will be created that actually

consist of proxies that will implement the discovered interface. Thus a client could now go ahead and get access to the bean by letting Spring simply autowire it.

Example 2-5. Using a Spring Data repository from a client

```
@Component
public class MyRepositoryClient {

  private final CustomerRepository repository;

  @Autowired
  public MyRepositoryClient(CustomerRepository repository) {
    Assert.notNull(repository);
    this.repository = repository;
  }

  …
}
```

With our `CustomerRepository` interface set up, we are ready to dive in and add some easy-to-declare query methods. A typical requirement might be to retrieve a `Customer` by its email address. To do so, we add the appropriate query method (Example 2-6).

Example 2-6. Declaring a query method

```
public interface CustomerRepository extends Repository<Customer, Long> {

  Customer findByEmailAddress(EmailAddress email);
}
```

The namespace element will now pick up the interface at container startup time and trigger the Spring Data infrastructure to create a Spring bean for it. The infrastructure will inspect the methods declared inside the interface and try to determine a query to be executed on method invocation. If you don't do anything more than declare the method, Spring Data will derive a query from its name. There are other options for query definition as well; you can read more about them in "Defining Query Methods" on page 16.

In Example 2-6, the query can be derived because we followed the naming convention of the domain object's properties. The `Email` part of the query method name actually refers to the `Customer` class's `emailAddress` property, and thus Spring Data will automatically derive `select C from Customer c where c.emailAddress = ?1` for the method declaration if you were using the JPA module. It will also check that you have valid property references inside your method declaration, and cause the container to fail to start on bootstrap time if it finds any errors. Clients can now simply execute the method, causing the given method parameters to be bound to the query derived from the method name and the query to be executed (Example 2-7).

Example 2-7. Executing a query method

```
@Component
public class MyRepositoryClient {

  private final CustomerRepository repository;

  …

  public void someBusinessMethod(EmailAddress email) {

    Customer customer = repository.findByEmailAddress(email);
  }
}
```

Defining Query Methods

Query Lookup Strategies

The interface we just saw had a simple query method declared. The method declaration was inspected by the infrastructure and parsed, and a store-specific query was derived eventually. However, as the queries become more complex, the method names would just become awkwardly long. For more complex queries, the keywords supported by the method parser wouldn't even suffice. Thus, the individual store modules ship with an `@Query` annotation, demonstrated in Example 2-8, that takes a query string in the store-specific query language and potentially allows further tweaks regarding the query execution.

Example 2-8. Manually defining queries using the @Query annotation

```
public interface CustomerRepository extends Repository<Customer, Long> {

  @Query("select c from Customer c where c.emailAddress = ?1")
  Customer findByEmailAddress(EmailAddress email);
}
```

Here we use JPA as an example and manually define the query that would have been derived anyway.

The queries can even be externalized into a properties file—*`$store`-named-queries.properties*, located in *META-INF*—where *`$store`* is a placeholder for *jpa*, *mongo*, *neo4j*, etc. The key has to follow the convention of *`$domainType.$methodName`*. Thus, to back our existing method with a externalized named query, the key would have to be `Customer.findByEmailAddress`. The `@Query` annotation is not needed if named queries are used.

Query Derivation

The query derivation mechanism built into the Spring Data repository infrastructure, shown in Example 2-9, is useful to build constraining queries over entities of the repository. We will strip the prefixes `findBy`, `readBy`, and `getBy` from the method and start parsing the rest of it. At a very basic level, you can define conditions on entity properties and concatenate them with `And` and `Or`.

Example 2-9. Query derivation from method names

```
public interface CustomerRepository extends Repository<Customer, Long> {

  List<Customer> findByEmailAndLastname(EmailAddress email, String lastname);
}
```

The actual result of parsing that method will depend on the data store we use. There are also some general things to notice. The expressions are usually property traversals combined with operators that can be concatenated. As you can see in Example 2-9, you can combine property expressions with `And` and `Or`. Beyond that, you also get support for various operators like `Between`, `LessThan`, `GreaterThan`, and `Like` for the property expressions. As the operators supported can vary from data store to data store, be sure to look at each store's corresponding chapter.

Property expressions

Property expressions can just refer to a direct property of the managed entity (as you just saw in Example 2-9). On query creation time, we already make sure that the parsed property is a property of the managed domain class. However, you can also define constraints by traversing nested properties. As seen above, `Customer`s have `Address`es with `ZipCode`s. In that case, a method name of:

```
List<Customer> findByAddressZipCode(ZipCode zipCode);
```

will create the property traversal `x.address.zipCode`. The resolution algorithm starts with interpreting the entire part (`AddressZipCode`) as a property and checks the domain class for a property with that name (with the first letter lowercased). If it succeeds, it just uses that. If not, it starts splitting up the source at the camel case parts from the right side into a head and a tail and tries to find the corresponding property (e.g., `AddressZip` and `Code`). If it finds a property with that head, we take the tail and continue building the tree down from there. Because in our case the first split does not match, we move the split point further to the left (from "`AddressZip`, `Code`" to "`Address`, `ZipCode`").

Although this should work for most cases, there might be situations where the algorithm could select the wrong property. Suppose our `Customer` class has an `addressZip` property as well. Then our algorithm would match in the first split, essentially choosing the wrong property, and finally fail (as the type of `addressZip` probably has no code

property). To resolve this ambiguity, you can use an underscore (_) inside your method name to manually define traversal points. So our method name would end up like so:

```
List<Customer> findByAddress_ZipCode(ZipCode zipCode);
```

Pagination and Sorting

If the number of results returned from a query grows significantly, it might make sense to access the data in chunks. To achieve that, Spring Data provides a pagination API that can be used with the repositories. The definition for what chunk of data needs to be read is hidden behind the `Pageable` interface alongside its implementation `PageRequest`. The data returned from accessing it page by page is held in a `Page`, which not only contains the data itself but also metainformation about whether it is the first or last page, how many pages there are in total, etc. To calculate this metadata, we will have to trigger a second query as well as the initial one.

We can use the pagination functionality with the repository by simply adding a `Pageable` as a method parameter. Unlike the others, this will not be bound to the query, but rather used to restrict the result set to be returned. One option is to have a return type of `Page`, which will restrict the results, but require another query to calculate the metainformation (e.g., the total number of elements available). Our other option is to use `List`, which will avoid the additional query but won't provide the metadata. If you don't need pagination functionality, but plain sorting only, add a `Sort` parameter to the method signature (see Example 2-10).

Example 2-10. Query methods using Pageable and Sort

```
Page<Customer> findByLastname(String lastname, Pageable pageable);

List<Customer> findByLastname(String lastname, Sort sort);

List<Customer> findByLastname(String lastname, Pageable pageable);
```

The first method allows you to pass a `Pageable` instance to the query method to dynamically add paging to your statically defined query. Sorting options can either be handed into the method by the `Sort` parameter explicitly, or embedded in the `PageRequest` value object, as you can see in Example 2-11.

Example 2-11. Using Pageable and Sort

```
Pageable pageable = new PageRequest(2, 10, Direction.ASC, "lastname", "firstname");
Page<Customer> result = findByLastname("Matthews", pageable);

Sort sort = new Sort(Direction.DESC, "Matthews");
List<Customer> result = findByLastname("Matthews", sort);
```

Defining Repositories

So far, we have seen repository interfaces with query methods derived from the method name or declared manually, depending on the means provided by the Spring Data module for the actual store. To derive these queries, we had to extend a Spring Data–specific marker interface: `Repository`. Apart from queries, there is usually quite a bit of functionality that you need to have in your repositories: the ability to store objects, to delete them, look them up by ID, return all entities stored, or access them page by page. The easiest way to expose this kind of functionality through the repository interfaces is by using one of the more advanced repository interfaces that Spring Data provides:

`Repository`
: A plain marker interface to let the Spring Data infrastructure pick up user-defined repositories

`CrudRepository`
: Extends `Repository` and adds basic persistence methods like saving, finding, and deleting entities

`PagingAndSortingRepositories`
: Extends `CrudRepository` and adds methods for accessing entities page by page and sorting them by given criteria

Suppose we want to expose typical CRUD operations for the `CustomerRepository`. All we need to do is change its declaration as shown in Example 2-12.

Example 2-12. CustomerRepository exposing CRUD methods

```
public interface CustomerRepository extends CrudRepository<Customer, Long> {

  List<Customer> findByEmailAndLastname(EmailAddress email, String lastname);
}
```

The `CrudRepository` interface now looks something like Example 2-13. It contains methods to save a single entity as well as an `Iterable` of entities, finder methods for a single entity or all entities, and `delete(…)` methods of different flavors.

Example 2-13. CrudRepository

```
public interface CrudRepository<T, ID extends Serializable> extends Repository<T, ID> {

  <S extends T> save(S entity);
  <S extends T> Iterable<S> save(Iterable<S> entities);

  T findOne(ID id);
  Iterable<T> findAll();

  void delete(ID id);
  void delete(T entity);
  void deleteAll();
}
```

Each of the Spring Data modules supporting the repository approach ships with an implementation of this interface. Thus, the infrastructure triggered by the namespace element declaration will not only bootstrap the appropriate code to execute the query methods, but also use an instance of the generic repository implementation class to back the methods declared in `CrudRepository` and eventually delegate calls to `save(…)`, `findAll()`, etc., to that instance. `PagingAndSortingRepository` (Example 2-14) now in turn extends `CrudRepository` and adds methods to allow handing instances of `Pageable` and `Sort` into the generic `findAll(…)` methods to actually access entities page by page.

Example 2-14. PagingAndSortingRepository

```
public interface PagingAndSortingRepository<T, ID extends Serializable>
  extends CrudRepository<T, ID> {

  Iterable<T> findAll(Sort sort);

  Page<T> findAll(Pageable pageable);
}
```

To pull that functionality into the `CustomerRepository`, you'd simply extend `PagingAndSortingRepository` instead of `CrudRepository`.

Fine-Tuning Repository Interfaces

As we've just seen, it's very easy to pull in chunks of predefined functionality by extending the appropriate Spring Data repository interface. The decision to implement this level of granularity was actually driven by the trade-off between the number of interfaces (and thus complexity) we would expose in the event that we had separator interfaces for all find methods, all save methods, and so on, versus the ease of use for developers.

However, there might be scenarios in which you'd like to expose only the reading methods (the R in CRUD) or simply prevent the delete methods from being exposed in your repository interfaces. Spring Data now allows you to tailor a custom base repository with the following steps:

1. Create an interface either extending `Repository` or annotated with `@RepositoryDefinition`.
2. Add the methods you want to expose to it and make sure they actually match the signatures of methods provided by the Spring Data base repository interfaces.
3. Use this interface as a base interface for the interface declarations for your entities.

To illustrate this, let's assume we'd like to expose only the `findAll(…)` method taking a `Pageable` as well as the save methods. The base repository interface would look like Example 2-15.

Example 2-15. Custom base repository interface

```
@NoRepositoryBean
public interface BaseRepository<T, ID extends Serializable> extends Repository<T, ID> {

  Iterable<T> findAll(Pageable sort);

  <S extends T> S save(S entity);

  <S extends T> S save(Iterable<S> entities);
}
```

Note that we additionally annotated the interface with `@NoRepositoryBean` to make sure the Spring Data repository infrastructure doesn't actually try to create a bean instance for it. Letting your `CustomerRepository` extend this interface will now expose exactly the API you defined.

It's perfectly fine to come up with a variety of base interfaces (e.g., a `ReadOnlyRepository` or a `SaveOnlyRepository`) or even a hierarchy of them depending on the needs of your project. We usually recommend starting with locally defined CRUD methods directly in the concrete repository for an entity and then moving either to the Spring Data–provided base repository interfaces or tailor-made ones if necessary. That way, you keep the number of artifacts naturally growing with the project's complexity.

Manually Implementing Repository Methods

So far we have seen two categories of methods on a repository: CRUD methods and query methods. Both types are implemented by the Spring Data infrastructure, either by a backing implementation or the query execution engine. These two cases will probably cover a broad range of data access operations you'll face when building applications. However, there will be scenarios that require manually implemented code. Let's see how we can achieve that.

We start by implementing just the functionality that actually needs to be implemented manually, and follow some naming conventions with the implementation class (as shown in Example 2-16).

Example 2-16. Implementing custom functionality for a repository

```
interface CustomerRepositoryCustom {

  Customer myCustomMethod(…);
}


class CustomerRepositoryImpl implements CustomerRepositoryCustom {

  // Potentially wire dependencies

  public Customer myCustomMethod(…) {
    // custom implementation code goes here
```

```
  }
}
```

Neither the interface nor the implementation class has to know anything about Spring Data. This works pretty much the way that you would manually implement code with Spring. The most interesting piece of this code snippet in terms of Spring Data is that the name of the implementation class follows the naming convention to suffix the core repository interface's (`CustomerRepository` in our case) name with `Impl`. Also note that we kept both the interface as well as the implementation class as *package private* to prevent them being accessed from outside the package.

The final step is to change the declaration of our original repository interface to extend the just-introduced one, as shown in Example 2-17.

Example 2-17. Including custom functionality in the CustomerRepository

```
public interface CustomerRepository extends CrudRepository<Customer, Long>,
                                             CustomerRepositoryCustom { … }
```

Now we have essentially pulled the API exposed in `CustomerRepositoryCustom` into our `CustomerRepository`, which makes it the central access point of the data access API for `Customer`s. Thus, client code can now call `CustomerRepository.myCustomMethod(…)`. But how does the implementation class actually get discovered and brought into the proxy to be executed eventually? The bootstrap process for a repository essentially looks as follows:

1. The repository interface is discovered (e.g., `CustomerRepository`).
2. We're trying to look up a bean definition with the name of the lowercase interface name suffixed by `Impl` (e.g., `customerRepositoryImpl`). If one is found, we'll use that.
3. If not, we scan for a class with the name of our core repository interface suffixed by `Impl` (e.g., `CustomerRepositoryImpl`, which will be picked up in our case). If one is found, we register this class as a Spring bean and use that.
4. The found custom implementation will be wired to the proxy configuration for the discovered interface and act as a potential target for method invocation.

This mechanism allows you to easily implement custom code for a dedicated repository. The suffix used for implementation lookup can be customized on the XML namespace element or an attribute of the repository enabling annotation (see the individual store chapters for more details on that). The reference documentation also contains some material on how to implement custom behavior to be applied to multiple repositories.

IDE Integration

As of version 3.0, the Spring Tool Suite (STS) provides integration with the Spring Data repository abstraction. The core area of support provided for Spring Data by STS is the query derivation mechanism for finder methods. The first thing it helps you with to

validate your derived query methods right inside the IDE so that you don't actually have to bootstrap an `ApplicationContext`, but can eagerly detect typos you introduce into your method names.

STS is a special Eclipse distribution equipped with a set of plug-ins to ease building Spring applications as much as possible. The tool can be downloaded from the project's website or installed into an plain Eclipse distribution by using the STS update site (based on Eclipse 3.8 or 4.2).

As you can see in Figure 2-1, the IDE detects that `Descrption` is not valid, as there is no such property available on the `Product` class. To discover these typos, it will analyze the `Product` domain class (something that bootstrapping the Spring Data repository infrastructure would do anyway) for properties and parse the method name into a property traversal tree. To avoid these kinds of typos as early as possible, STS's Spring Data support offers code completion for property names, criteria keywords, and concatenators like `And` and `Or` (see Figure 2-2).

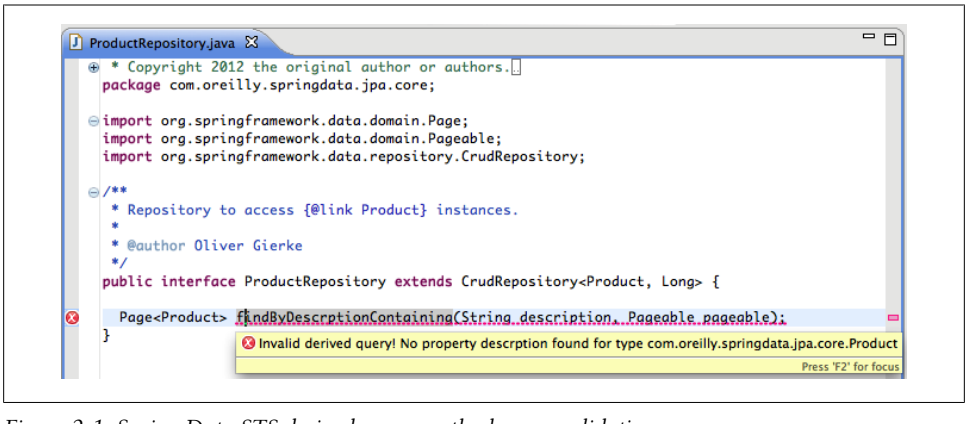


Figure 2-1. Spring Data STS derived query method name validation

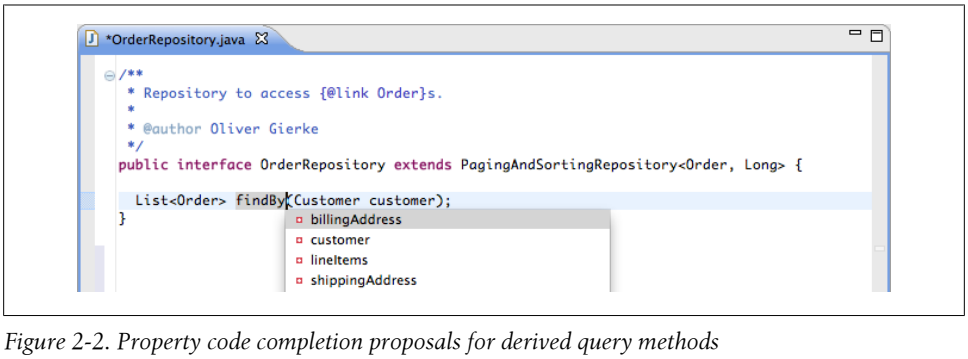


Figure 2-2. Property code completion proposals for derived query methods

The `Order` class has a few properties that you might want to refer to. Assuming we'd like to traverse the `billingAddress` property, another Cmd+Space (or Ctrl+Space on Windows) would trigger a nested property traversal that proposes nested properties, as well as keywords matching the type of the property traversed so far (Figure 2-3). Thus, `String` properties would additionally get `Like` proposed.

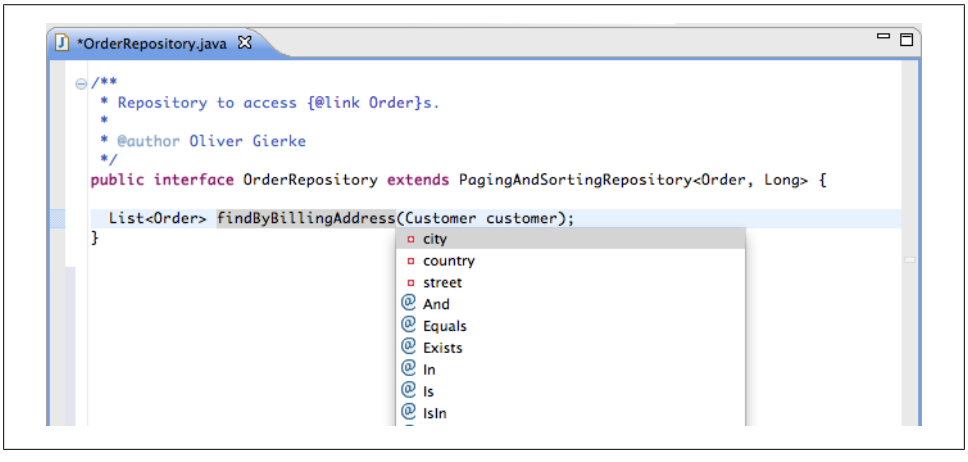


Figure 2-3. Nested property and keyword proposals

To put some icing on the cake, the Spring Data STS will make the repositories first-class citizens of your IDE navigator, marking them with the well-known Spring bean symbol. Beyond that, the Spring Elements node in the navigator will contain a dedicated Spring Data Repositories node to contain all repositories found in your application's configuration (see Figure 2-4).

As you can see, you can discover the repository interfaces at a quick glance and trace which configuration element they actually originate from.

IntelliJ IDEA

Finally, with the JPA support enabled, IDEA offers repository finder method completion derived from property names and the available keyword, as shown in Figure 2-5.

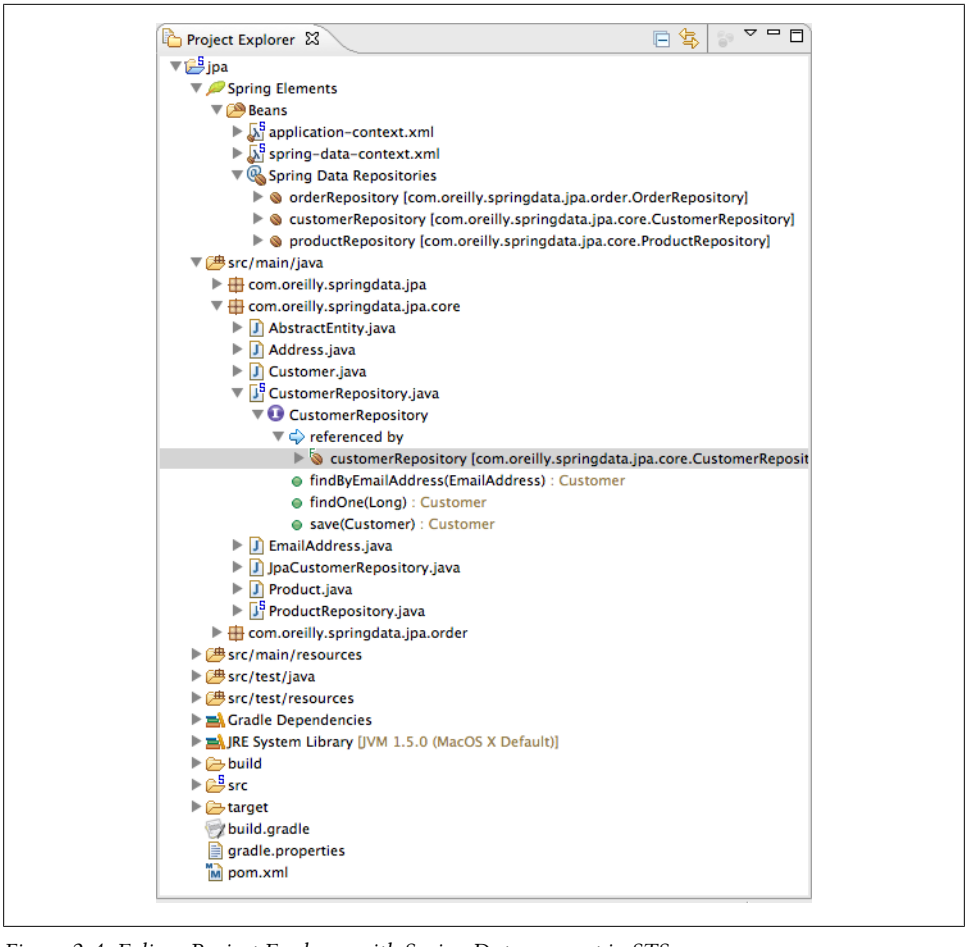


Figure 2-4. Eclipse Project Explorer with Spring Data support in STS

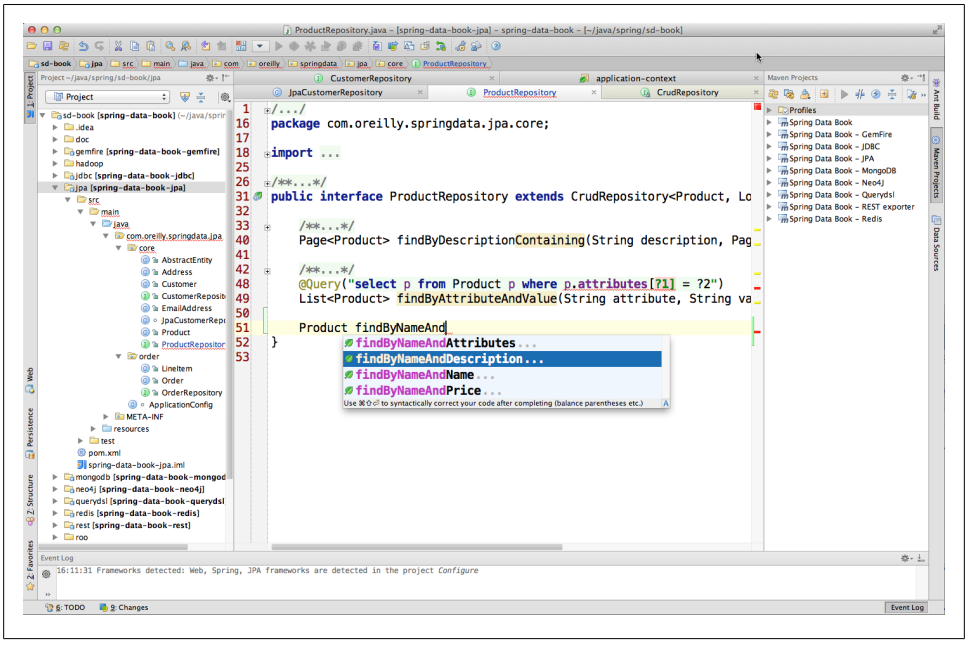


Figure 2-5. Finder method completion in IDEA editor

CHAPTER 3

Type-Safe Querying Using Querydsl

Writing queries to access data is usually done using Java `String`s. The query languages of choice have been SQL for JDBC as well as HQL/JPQL for Hibernate/JPA. Defining the queries in plain `String`s is powerful but quite error-prone, as it's very easy to introduce typos. Beyond that, there's little coupling to the actual query source or sink, so column references (in the JDBC case) or property references (in the HQL/JPQL context) become a burden in maintenance because changes in the table or object model cannot be reflected in the queries easily.

The Querydsl project tries to tackle this problem by providing a fluent API to define these queries. The API is derived from the actual table or object model but is highly store- and model-agnostic at the same time, so it allows you to create and use the query API for a variety of stores. It currently supports JPA, Hibernate, JDO, native JDBC, Lucene, Hibernate Search, and MongoDB. This versatility is the main reason why the Spring Data project integrates with Querydsl, as Spring Data also integrates with a variety of stores. The following sections will introduce you to the Querydsl project and its basic concepts. We will go into the details of the store-specific support in the store-related chapters later in this book.

Introduction to Querydsl

When working with Querydsl, you will usually start by deriving a metamodel from your domain classes. Although the library can also use plain `String` literals, creating the metamodel will unlock the full power of Querydsl, especially its type-safe property and keyword references. The derivation mechanism is based on the Java 6 Annotation Processing Tool (APT), which allows for hooking into the compiler and processing the sources or even compiled classes. For details, read up on that topic in "Generating the Query Metamodel" on page 30. To kick things off, we need to define a domain class like the one shown in Example 3-1. We model our `Customer` with a few primitive and nonprimitive properties.

Example 3-1. The Customer domain class

```
@QueryEntity
public class Customer extends AbstractEntity {

  private String firstname, lastname;
  private EmailAddress emailAddress;
  private Set<Address> addresses = new HashSet<Address>();

  …
}
```

Note that we annotate the class with `@QueryEntity`. This is the default annotation, from which the Querydsl annotation processor generates the related query class. When you're using the integration with a particular store, the APT processor will be able to recognize the store-specific annotations (e.g., `@Entity` for JPA) and use them to derive the query classes. As we're not going to work with a store for this introduction and thus cannot use a store-specific mapping annotation, we simply stick with `@QueryEntity`. The generated Querydsl query class will now look like Example 3-2.

Example 3-2. The Querydsl generated query class

```
@Generated("com.mysema.query.codegen.EntitySerializer")
public class QCustomer extends EntityPathBase<Customer> {

  public static final QCustomer customer = new QCustomer("customer");
  public final QAbstractEntity _super = new QAbstractEntity(this);

  public final NumberPath<Long> id = _super.id;
  public final StringPath firstname = createString("firstname");
  public final StringPath lastname = createString("lastname");
  public final QEmailAddress emailAddress;

  public final SetPath<Address, QAddress> addresses =
         this.<Address, QAddress>createSet("addresses", Address.class, QAddress.class);

  …
}
```

You can find these classes in the *target/generated-sources/queries* folder of the module's sample project. The class exposes public `Path` properties and references to other query classes (e.g., `QEmailAddress`). This enables your IDE to list the available paths for which you might want to define predicates during code completion. You can now use these `Path` expressions to define reusable predicates, as shown in Example 3-3.

Example 3-3. Using the query classes to define predicates

```
QCustomer customer = QCustomer.customer;

BooleanExpression idIsNull = customer.id.isNull();
BooleanExpression lastnameContainsFragment = customer.lastname.contains("thews");
BooleanExpression firstnameLikeCart = customer.firstname.like("Cart");
```

```
EmailAddress reference = new EmailAddress("dave@dmband.com");
BooleanExpression isDavesEmail = customer.emailAddress.eq(reference);
```

We assign the static `QCustomer.customer` instance to the `customer` variable to be able to concisely refer to its property paths. As you can see, the definition of a predicate is clean, concise, and—most importantly—type-safe. Changing the domain class would cause the query metamodel class to be regenerated. Property references that have become invalidated by this change would become compiler errors and thus give us hints to places that need to be adapted. The methods available on each of the `Path` types take the type of the `Path` into account (e.g., the `like(…)` method makes sense only on `String` properties and thus is provided only on those).

Because predicate definitions are so concise, they can easily be used inside a method declaration. On the other hand, we can easily define predicates in a reusable manner, building up atomic predicates and combining them with more complex ones by using concatenating operators like `And` and `Or` (see Example 3-4).

Example 3-4. Concatenating atomic predicates

```
QCustomer customer = QCustomer.customer;

BooleanExpression idIsNull = customer.id.isNull();

EmailAddress reference = new EmailAddress("dave@dmband.com");
BooleanExpression isDavesEmail = customer.emailAddress.eq(reference);

BooleanExpression idIsNullOrIsDavesEmail = idIsNull.or(isDavesEmail);
```

We can use our newly written predicates to define a query for either a particular store or plain collections. As the support for store-specific query execution is mainly achieved through the Spring Data repository abstraction, have a look at "Integration with Spring Data Repositories" on page 32. We'll use the feature to query collections as an example now to keep things simple. First, we set up a variety of `Products` to have something we can filter, as shown in Example 3-5.

Example 3-5. Setting up Products

```
Product macBook = new Product("MacBook Pro", "Apple laptop");
Product iPad = new Product("iPad", "Apple tablet");
Product iPod = new Product("iPod", "Apple MP3 player");
Product turntable = new Product("Turntable", "Vinyl player");

List<Product> products = Arrays.asList(macBook, iPad, iPod, turntable);
```

Next, we can use the Querydsl API to actually set up a query against the collection, which is some kind of filter on it (Example 3-6).

Example 3-6. Filtering Products using Querydsl predicates

```
QProduct $ = QProduct.product;
List<Product> result = from($, products).where($.description.contains("Apple")).list($);
```

```
assertThat(result, hasSize(3));
assertThat(result, hasItems(macBook, iPad, iPod));
```

We're setting up a Querydsl `Query` using the `from(…)` method, which is a static method on the `MiniAPI` class of the *querydsl-collections* module. We hand it an instance of the query class for `Product` as well as the source collection. We can now use the `where(…)` method to apply predicates to the source list and execute the query using one of the `list(…)` methods (Example 3-7). In our case, we'd simply like to get back the `Product` instances matching the defined predicate. Handing `$.description` into the `list(…)` method would allow us to project the result onto the product's description and thus get back a collection of `String`s.

Example 3-7. Filtering Products using Querydsl predicates (projecting)

```
QProduct $ = QProduct.product;
BooleanExpression descriptionContainsApple = $.description.contains("Apple");
List<String> result = from($, products).where(descriptionContainsApple).list($.name);

assertThat(result, hasSize(3));
assertThat(result, hasItems("MacBook Pro", "iPad", "iPod"));
```

As we have discovered, Querydsl allows us to define entity predicates in a concise and easy way. These can be generated from the mapping information for a variety of stores as well as for plain Java classes. Querydsl's API and its support for various stores allows us to generate predicates to define queries. Plain Java collections can be filtered with the very same API.

Generating the Query Metamodel

As we've just seen, the core artifacts with Querydsl are the query metamodel classes. These classes are generated via the Annotation Processing Toolkit, part of the `javac` Java compiler in Java 6. The APT provides an API to programmatically inspect existing Java source code for certain annotations, and then call functions that in turn generate Java code. Querydsl uses this mechanism to provide special APT processor implementation classes that inspect annotations. Example 3-1 used Querydsl-specific annotations like `@QueryEntity` and `@QueryEmbeddable`. If we already have domain classes mapped to a store supported by Querydsl, then generating the metamodel classes will require no extra effort. The core integration point here is the annotation processor you hand to the Querydsl APT. The processors are usually executed as a build step.

Build System Integration

To integrate with Maven, Querydsl provides the *maven-apt-plugin*, with which you can configure the actual processor class to be used. In Example 3-8, we bind the `process` goal to the `generate-sources` phase, which will cause the configured processor class to

inspect classes in the *src/main/java* folder. To generate metamodel classes for classes in the test sources (*src/test/java*), add an execution of the `test-process` goal to the `generate-test-sources` phase.

Example 3-8. Setting up the Maven APT plug-in

```
<project …>
  <build>
    <plugins>
      <plugin>
        <groupId>com.mysema.maven</groupId>
        <artifactId>maven-apt-plugin</artifactId>
        <version>1.0.2</version>
        <executions>
          <execution>
            <goals>
              <goal>process</goal>
            </goals>
            <phase>generate-sources</phase>
            <configuration>
              <outputDirectory>target/generated-sources/java</outputDirectory>
              <processor><!-- fully-qualified processor class name --></processor>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Supported Annotation Processors

Querydsl ships with a variety of APT processors to inspect different sets of annotations and generate the metamodel classes accordingly.

`QuerydslAnnotationProcessor`
: The very core annotation processor inspects Querydsl-specific annotations like `@QueryEntity` and `@QueryEmbeddable`.

`JPAAnnotationProcessor`
: Inspects `javax.persistence` annotations, such as `@Entity` and `@Embeddable`.

`HibernateAnnotationProcessor`
: Similar to the JPA processor but adds support for Hibernate-specific annotations.

`JDOAnnotationProcessor`
: Inspects JDO annotations, such as `@PersistenceCapable` and `@EmbeddedOnly`.

`MongoAnnotationProcessor`
: A Spring Data–specific processor inspecting the `@Document` annotation. Read more on this in "The Mapping Subsystem" on page 83.

Querying Stores Using Querydsl

Now that we have the query classes in place, let's have a look at how we can use them to actually build queries for a particular store. As already mentioned, Querydsl provides integration modules for a variety of stores that offer a nice and consistent API to create query objects, apply predicates defined via the generated query metamodel classes, and eventually execute the queries.

The JPA module, for example, provides a `JPAQuery` implementation class that takes an `EntityManager` and provides an API to apply predicates before execution; see Example 3-9.

Example 3-9. Using Querydsl JPA module to query a relational store

```
EntityManager entityManager = … // obtain EntityManager
JPAQuery query = new JPAQuery(entityManager);

QProduct $ = QProduct.product;
List<Product> result = query.from($).where($.description.contains("Apple")).list($);

assertThat(result, hasSize(3));
assertThat(result, hasItems(macBook, iPad, iPod));
```

If you remember Example 3-6, this code snippet doesn't look very different. In fact, the only difference here is that we use the `JPAQuery` as the base, whereas the former example used the collection wrapper. So you probably won't be too surprised to see that there's not much difference in implementing this scenario for a MongoDB store (Example 3-10).

Example 3-10. Using Querydsl MongoDB module with Spring Data MongoDB

```
MongoOperations operations = … // obtain MongoOperations
MongodbQuery query = new SpringDataMongodbQuery(operations, Product.class);

QProduct $ = QProduct.product;
List<Product> result = query.where($.description.contains("Apple").list();

assertThat(result, hasSize(3));
assertThat(result, hasItems(macBook, iPad, iPod));
```

Integration with Spring Data Repositories

As you just saw, the execution of queries with Querydsl generally consists of three major steps:

1. Setting up a store-specific query instance
2. Applying a set of filter predicates to it
3. Executing the query instance, potentially applying projections to it

Two of these steps can be considered boilerplate, as they will usually result in very similar code being written. On the other hand, the Spring Data repository tries to help users reduce the amount of unnecessary code; thus, it makes sense to integrate the repository extraction with Querydsl.

Executing Predicates

The core of the integration is the `QueryDslPredicateExecutor` interface ,which specifies the API that clients can use to execute Querydsl predicates in the flavor of the CRUD methods provided through `CrudRepository`. See Example 3-11.

Example 3-11. The QueryDslPredicateExecutor interface

```
public interface QueryDslPredicateExecutor<T> {

  T findOne(Predicate predicate);

  Iterable<T> findAll(Predicate predicate);
  Iterable<T> findAll(Predicate predicate, OrderSpecifier<?>... orders);

  Page<T> findAll(Predicate predicate, Pageable pageable);
  long count(Predicate predicate);
}
```

Currently, Spring Data JPA and MongoDB support this API by providing implementation classes implementing the QueryDslPredicateExecutor interface shown in Example 3-11. To expose this API through your repository interfaces, let it extend `QueryDslPredicateExecutor` in addition to `Repository` or any of the other available base interfaces (see Example 3-12).

Example 3-12. The CustomerRepository interface extending QueryDslPredicateExecutor

```
public interface CustomerRepository extends Repository<Customer, Long>,
                                            QueryDslPredicateExecutor<Customer> {

  …
}
```

Extending the interface will have two important results: the first—and probably most obvious—is that it pulls in the API and thus exposes it to clients of `CustomerRepository`. Second, the Spring Data repository infrastructure will inspect each repository interface found to determine whether it extends `QueryDslPredicateExecutor`. If it does and Querydsl is present on the classpath, Spring Data will select a special base class to back the repository proxy that generically implements the API methods by creating a store-specific query instance, bind the given predicates, potentially apply pagination, and eventually execute the query.

Manually Implementing Repositories

The approach we have just seen solves the problem of generically executing queries for the domain class managed by the repository. However, you cannot execute updates or deletes through this mechanism or manipulate the store-specific query instance. This is actually a scenario that plays nicely into the feature of *repository abstraction*, which allows you to selectively implement methods that need hand-crafted code (see "Manually Implementing Repository Methods" on page 21 for general details on that topic). To ease the implementation of a custom repository extension, we provide store-specific base classes. For details on that, check out the sections "Repository Querydsl Integration" on page 51 and "Mongo Querydsl Integration" on page 99.

PART II

Relational Databases

CHAPTER 4
JPA Repositories

The Java Persistence API (JPA) is the standard way of persisting Java objects into relational databases. The JPA consists of two parts: a mapping subsystem to map classes onto relational tables as well as an `EntityManager` API to access the objects, define and execute queries, and more. JPA abstracts a variety of implementations such as Hibernate, EclipseLink, OpenJpa, and others. The Spring Framework has always offered sophisticated support for JPA to ease repository implementations. The support consists of helper classes to set up an `EntityManagerFactory`, integrate with the Spring transaction abstraction, and translate JPA-specific exceptions into Spring's `DataAccessException` hierarchy.

The Spring Data JPA module implements the Spring Data Commons repository abstraction to ease the repository implementations even more, making a manual implementation of a repository obsolete in most cases. For a general introduction to the repository abstraction, see Chapter 2. This chapter will take you on a guided tour through the general setup and features of the module.

The Sample Project

Our sample project for this chapter consists of three packages: the *com.oreilly.springdata.jpa* base package plus a *core* and an *order* subpackage. The base package contains a Spring JavaConfig class to configure the Spring container using a plain Java class instead of XML. The two other packages contain our domain classes and repository interfaces. As the name suggests, the core package contains the very basic abstractions of the domain model: technical helper classes like `AbstractEntity`, but also domain concepts like an `EmailAddress`, an `Address`, a `Customer`, and a `Product`. Next, we have the *orders* package, which implements actual order concepts built on top of the foundational ones. So we'll find the `Order` and its `LineItem`s here. We will have a closer look at each of these classes in the following paragraphs, outlining their purpose and the way they are mapped onto the database using JPA mapping annotations.

The very core base class of all entities in our domain model is `AbstractEntity` (see Example 4-1). It's annotated with `@MappedSuperclass` to express that it is not a managed entity class on its own but rather will be extended by entity classes. We declare an `id` of type `Long` here and instruct the persistence provider to automatically select the most appropriate strategy for autogeneration of primary keys. Beyond that, we implement `equals(…)` and `hashCode()` by inspecting the `id` property so that entity classes of the same type with the same `id` are considered equal. This class contains the main technical artifacts to persist an entity so that we can concentrate on the actual domain properties in the concrete entity classes.

Example 4-1. The AbstractEntity class

```
@MappedSuperclass
public class AbstractEntity {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;

  @Override
  public boolean equals(Object obj) { … }

  @Override
  public int hashCode() { … }
}
```

Let's proceed with the very simple `Address` domain class. As Example 5-2 shows, it is a plain `@Entity` annotated class and simply consists of three `String` properties. Because they're all basic ones, no additional annotations are needed, and the persistence provider will automatically map them into table columns. If there were demand to customize the names of the columns to which the properties would be persisted, you could use the `@Column` annotation.

Example 4-2. The Address domain class

```
@Entity
public class Address extends AbstractEntity {

  private String street, city, country;
}
```

The `Address`es are referred to by the `Customer` entity. `Customer` contains quite a few other properties (e.g., the primitive ones `firstname` and `lastname`). They are mapped just like the properties of `Address` that we have just seen. Every `Customer` also has an email address represented through the `EmailAddress` class (see Example 4-3).

Example 4-3. The EmailAddress domain class

```
@Embeddable
public class EmailAddress {
```

```
  private static final String EMAIL_REGEX = …;
  private static final Pattern PATTERN = Pattern.compile(EMAIL_REGEX);

  @Column(name = "email")
  private String emailAddress;

  public EmailAddress(String emailAddress) {
    Assert.isTrue(isValid(emailAddress), "Invalid email address!");
    this.emailAddress = emailAddress;
  }

  protected EmailAddress() { }

  public boolean isValid(String candidate) {
    return PATTERN.matcher(candidate).matches();
  }
}
```

This class is a value object, as defined in Eric Evans's book *Domain Driven Design* [Evans03]. Value objects are usually used to express domain concepts that you would naively implement as a primitive type (a `string` in this case) but that allow implementing domain constraints inside the value object. Email addresses have to adhere to a specific format; otherwise, they are not valid email addresses. So we actually implement the format check through some regular expression and thus prevent an `EmailAddress` instance from being instantiated if it's invalid.

This means that we can be sure to have a valid email address if we deal with an instance of that type, and we don't have to have some component validate it for us. In terms of persistence mapping, the `EmailAddress` class is an `@Embeddable,` which will cause the persistence provider to flatten out all properties of it into the table of the surrounding class. In our case, it's just a single column for which we define a custom name: `email`.

As you can see, we need to provide an empty constructor for the JPA persistence provider to be able to instantiate `EmailAddress` objects via reflection (Example 5-4). This is a significant shortcoming because you effectively cannot make the `emailAddress` a final one or assert make sure it is not `null`. The Spring Data mapping subsystem used for the NoSQL store implementations does not impose that need onto the developer. Have a look at "The Mapping Subsystem" on page 83 to see how a stricter implementation of the value object can be modeled in MongoDB, for example.

Example 4-4. The Customer domain class

```
@Entity
public class Customer extends AbstractEntity{

  private String firstname, lastname;

  @Column(unique = true)
  private EmailAddress emailAddress;
```

```
  @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
  @JoinColumn(name = "customer_id")
  private Set<Address> addresses;
}
```

We use the `@Column` annotation on the email address to make sure a single email address cannot be used by multiple customers so that we are able to look up customers uniquely by their email address. Finally we declare the `Customer` having a set of `Addresses`. This property deserves deeper attention, as there are quite a few things we define here.

First, and in general, we use the `@OneToMany` annotation to specify that one `Customer` can have multiple `Addresses`. Inside this annotation, we set the cascade type to `CascadeType.ALL` and also activate orphan removal for the addresses. This has a few consequences. For example, whenever we initially persist, update, or delete a customer, the `Addresses` will be persisted, updated, or deleted as well. Thus, we don't have to persist an `Address` instance up front or take care of removing all `Addresses` whenever we delete a `Customer`; the persistence provider will take care of that. Note that this is not a database-level cascade but rather a cascade managed by your JPA persistence provider. Beyond that, setting the orphan removal flag to `true` will take care of deleting `Addresses` from that database if they are removed from the collection.

All this results in the `Address` life cycle being controlled by the `Customer`, which makes the relationship a classical composition. Plus, in domain-driven design (DDD) terminology, the `Customer` qualifies as aggregate root because it controls persistence operations and constraints for itself as well as other entities. Finally, we use `@JoinColumn` with the `addresses` property, which causes the persistence provider to add another column to the table backing the `Address` object. This additional column will then be used to refer to the `Customer` to allow joining the tables. If we had left out the additional annotation, the persistence provider would have created a dedicated join table.

The final piece of our core package is the `Product` (Example 4-5). Just as with the other classes discussed, it contains a variety of basic properties, so we don't need to add annotations to get them mapped by the persistence provider. We add only the `@Column` annotation to define the name and price as mandatory properties. Beyond that, we add a `Map` to store additional attributes that might differ from `Product` to `Product`.

Example 4-5. The Product domain class

```
@Entity
public class Product extends AbstractEntity {

  @Column(nullable = false)
  private String name;
  private String description;

  @Column(nullable = false)
  private BigDecimal price;
```

```
  @ElementCollection
  private Map<String, String> attributes = new HashMap<String, String>();
}
```

Now we have everything in place to build a basic customer relation management (CRM) or inventory system. Next, we're going to add abstractions that allow us to implement orders for `Product`s held in the system. First, we introduce a `LineItem` that captures a reference to a `Product` alongside the amount of products as well as the price at which the product was bought. We map the `Product` property using a `@ManyToOne` annotation that will actually be turned into a `product_id` column in the `LineItem` table pointing to the `Product` (see Example 4-6).

Example 4-6. The LineItem domain class

```
@Entity
public class LineItem extends AbstractEntity {

  @ManyToOne
  private Product product;

  @Column(nullable = false)
  private BigDecimal price;
  private int amount;
}
```

The final piece to complete the jigsaw puzzle is the `Order` entity, which is basically a pointer to a `Customer`, a shipping `Address`, a billing `Address`, and the `LineItem`s actually ordered (Example 4-7). The mapping of the line items is the very same as we already saw with `Customer` and `Address`. The `Order` will automatically cascade persistence operations to the `LineItem` instances. Thus, we don't have to manage the persistence life cycle of the `LineItem`s separately. All other properties are many-to-one relationships to concepts already introduced. Note that we define a custom table name to be used for `Order`s because `Order` itself is a reserved keyword in most databases; thus, the generated SQL to create the table as well as all SQL generated for queries and data manipulation would cause exceptions when executing.

Example 4-7. The Order domain class

```
@Entity
@Table(name = "Orders")
public class Order extends AbstractEntity {

  @ManyToOne(optional = false)
  private Customer customer;
  @ManyToOne
  private Address billingAddress;

  @ManyToOne(optional = false, cascade = CascadeType.ALL)
  private Address shippingAddress;
```

```
  @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
  @JoinColumn(name = "order_id")
  private Set<LineItem>;

  …

  public Order(Customer customer, Address shippingAddress,
    Address billingAddress) {

    Assert.notNull(customer);
    Assert.notNull(shippingAddress);

    this.customer = customer;
    this.shippingAddress = shippingAddress.getCopy();
    this.billingAddress = billingAddress == null ? null :
      billingAddress.getCopy();
  }
}
```

A final aspect worth noting is that the constructor of the `Order` class does a defensive copy of the shipping and billing address. This is to ensure that changes to the `Address` instance handed into the method do not propagate into already existing orders. If we didn't create the copy, a customer changing her `Address` data later on would also change the `Address` on all of her `Order`s made to that `Address` as well.

The Traditional Approach

Before we start, let's look at how Spring Data helps us implement the data access layer for our domain model, and discuss how we'd implement the data access layer the traditional way. You'll find the sample implementation and client in the sample project annotated with additional annotations like `@Profile` (for the implementation) as well as `@ActiveProfile` (in the test case). This is because the Spring Data repositories approach will create an instance for the `CustomerRepository`, and we'll have one created for our manual implementation as well. Thus, we use the Spring profiles mechanism to bootstrap the traditional implementation for only the single test case. We don't show these annotations in the sample code because they would not have actually been used if you implemented the entire data access layer the traditional way.

To persist the previously shown entities using plain JPA, we now create an interface and implementation for our repositories, as shown in Example 4-8.

Example 4-8. Repository interface definition for Customers

```
public interface CustomerRepository {

  Customer save(Customer account);

  Customer findByEmailAddress(EmailAddress emailAddress);
}
```

So we declare a method `save(…)` to be able to store accounts, and a query method to find all accounts that are assigned to a given customer by his email address. Let's see what an implementation of this repository would look like if we implemented it on top of plain JPA (Example 4-9).

Example 4-9. Traditional repository implementation for Customers

```
@Repository
@Transactional(readOnly = true)
class JpaCustomerRepository implements CustomerRepository {

  @PersistenceContext
  private EntityManager em;

  @Override
  @Transactional
  public Customer save(Customer customer) {

    if (customer.getId() == null) {
      em.persist(customer);
      return customer;
    } else {
      return em.merge(customer);
    }
  }

  @Override
  public Customer findByEmailAddress(EmailAddress emailAddress) {

    TypedQuery<Customer> query = em.createQuery(
      "select c from Customer c where c.emailAddress = :emailAddress", Customer.class);
    query.setParameter("emailAddress", emailAddress);

    return query.getSingleResult();
  }
}
```

The implementation class uses a JPA `EntityManager`, which will get injected by the Spring container due to the JPA `@PersistenceContext` annotation. The class is annotated with `@Repository` to enable exception translation from JPA exceptions to Spring's `DataAccessException` hierarchy. Beyond that, we use `@Transactional` to make sure the `save(…)` operation is running in a transaction and to allow setting the `readOnly` flag (at the class level) for `findByEmailAddress(…)`. This helps optimize performance inside the persistence provider as well as on the database level.

Because we want to free the clients from the decision of whether to call `merge(…)` or `persist(…)` on the `EntityManager`, we use the `id` field of the `Customer` to specify whether we consider a `Customer` object as new or not. This logic could, of course, be extracted into a common repository superclass, as we probably don't want to repeat this code for every domain object–specific repository implementation. The query method is quite straightforward as well: we create a query, bind a parameter, and execute the query to

get a result. It's almost *so* straightforward that you could regard the implementation code as boilerplate. With a little bit of imagination, we can derive an implementation from the method signature: we expect a single customer, the query is quite close to the method name, and we simply bind the method parameter to it. So, as you can see, there's room for improvement.

Bootstrapping the Sample Code

We now have our application components in place, so let's get them up and running inside a Spring container. To do so, we have to do two things: first, we need to configure the general JPA infrastructure (i.e., a `DataSource` connecting to a database as well as a JPA `EntityManagerFactory`). For the former we will use HSQL, a database that supports being run in-memory. For the latter we will choose Hibernate as the persistence provider. You can find the dependency setup in the *pom.xml* file of the sample project. Second, we need to set up the Spring container to pick up our repository implementation and create a bean instance for it. In Example 4-10, you see a Spring JavaConfig configuration class that will achieve the steps just described.

Example 4-10. Spring JavaConfig configuration

```
@Configuration
@ComponentScan
@EnableTransactionManagement
class ApplicationConfig {

  @Bean
  public DataSource dataSource() {
    EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
    return builder.setType(EmbeddedDatabaseType.HSQL).build();
  }

  @Bean
  public LocalContainerEntityManagerFactoryBean entityManagerFactory() {

    HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
    vendorAdapter.setDatabase(Database.HSQL);
    vendorAdapter.setGenerateDdl(true);

    LocalContainerEntityManagerFactoryBean factory =
      new LocalContainerEntityManagerFactoryBean();
    factory.setJpaVendorAdapter(vendorAdapter);
    factory.setPackagesToScan(getClass().getPackage().getName());
    factory.setDataSource(dataSource());

    return factory;
  }

  @Bean
  public PlatformTransactionManager transactionManager() {
    JpaTransactionManager txManager = new JpaTransactionManager();
    txManager.setEntityManagerFactory(entityManagerFactory());
```

```
    return txManager;
  }
}
```

The `@Configuration` annotation declares the class as a Spring JavaConfig configuration class. The `@ComponentScan` instructs Spring to scan the package of the `ApplicationConfig` class and all of its subpackages for Spring components (classes annotated with `@Service`, `@Repository`, etc.). `@EnableTransactionManagement` activates Spring-managed transactions at methods annotated with `@Transactional`.

The methods annotated with `@Bean` now declare the following infrastructure components: `dataSource()` sets up an embedded data source using Spring's embedded database support. This allows you to easily set up various in-memory databases for testing purposes with almost no configuration effort. We choose HSQL here (other options are H2 and Derby). On top of that, we configure an `EntityManagerFactory`. We use a new Spring 3.1 feature that allows us to completely abstain from creating a *persistence.xml* file to declare the entity classes. Instead, we use Spring's classpath scanning feature through the `packagesToScan` property of the `LocalContainerEntityManagerFactoryBean`. This will trigger Spring to scan for classes annotated with `@Entity` and `@MappedSuperclass` and automatically add those to the JPA `PersistenceUnit`.

The same configuration defined in XML looks something like Example 4-11.

Example 4-11. XML-based Spring configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:jdbc="http://www.springframework.org/schema/jdbc"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx.xsd
                           http://www.springframework.org/schema/jdbc
                           http://www.springframework.org/schema/jdbc/spring-jdbc.xsd">

  <context:componen-scan base-package="com.oreilly.springdata.jpa" />

  <tx:annotation-driven />

  <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
  </bean>

  <bean id="entityManagerFactory"
        class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="packagesToScan" value="com.oreilly.springdata.jpa" />
```

```
    <property name="jpaVendorAdapter">
      <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
        <property name="database" value="HSQL" />
        <property name="generateDdl" value="true" />
      </bean>
    </property>
  </bean>

  <jdbc:embedded-database id="dataSource" type="HSQL" />

</beans>
```

The `<jdbc:embedded-database />` at the very bottom of this example creates the in-memory `Datasource` using HSQL. The declaration of the `LocalContainerEntityManagerFactoryBean` is analogous to the declaration in code we've just seen in the JavaConfig case (Example 4-10). On top of that, we declare the `JpaTransactionManager` and finally activate annotation-based transaction configuration and component scanning for our base package. Note that the XML configuration in Example 4-11 is slightly different from the one you'll find in the *META-INF/spring/application-context.xml* file of the sample project. This is because the sample code is targeting the Spring Data JPA-based data access layer implementation, which renders some of the configuration just shown obsolete.

The sample application class creates an instance of an `AnnotationConfigApplicationContext`, which takes a Spring JavaConfig configuration class to bootstrap application components (Example 4-12). This will cause the infrastructure components declared in our `ApplicationConfig` configuration class and our annotated repository implementation to be discovered and instantiated. Thus, we can access a Spring bean of type `CustomerRepository`, create a customer, store it, and look it up by its email address.

Example 4-12. Bootstrapping the sample code

```
@RunWith(SpringJunit4ClassRunner.class)
@ContextConfiguration(classes = ApplicationConfig.class)
class CustomerRepositoryIntegrationTests {

  @Autowired
  CustomerRepository customerRepository;

  @Test
  public void savesAndFindsCustomerByEmailAddress {

    Customer dave = new Customer("Dave", "Matthews");
    dave.setEmailAddress("dave@dmband.com");

    Customer result = repository.save(dave);
    Assert.assertThat(result.getId(), is(nonNullValue()));

    result = repository.findByEmailAddress("dave@dmband.com");
    Assert.assertThat(result, is(dave));
  }
}
```

Using Spring Data Repositories

To enable the Spring data repositories, we must make the repository interfaces discoverable by the Spring Data repository infrastructure. We do so by letting our `CustomerRepository` extend the Spring Data Repository marker interface. Beyond that, we keep the declared persistence methods we already have. See Example 4-13.

Example 4-13. Spring Data CustomerRepository interface

```
public interface CustomerRepository extends Repository<Customer, Long> {

  Customer save(Account account);

  Customer findByEmailAddress(String emailAddress);
}
```

The `save(…)` method will be backed by the generic `SimpleJpaRepository` class that implements all CRUD methods. The query method we declared will be backed by the generic query derivation mechanism, as described in "Query Derivation" on page 17. The only thing we now have to add to the Spring configuration is a way to activate the Spring Data repository infrastructure, which we can do in either XML or JavaConfig. For the JavaConfig way of configuration, all you need to do is add the `@EnableJpaRepositories` annotation to your configuration class. We remove the `@ComponentScan` annotation be removed for our sample because we don't need to look up the manual implementation anymore. The same applies to `@EnableTransactionManagement`. The Spring Data repository infrastructure will automatically take care of the method calls to repositories taking part in transactions. For more details on transaction configuration, see "Transactionality" on page 50. We'd probably still keep these annotations around were we building a more complete application. We remove them for now to prevent giving the impression that they are necessary for the sole data access setup. Finally, the header of the `ApplicationConfig` class looks something like Example 4-14.

Example 4-14. Enabling Spring Data repositories using JavaConfig

```
@Configuration
@EnableJpaRepositories
class ApplicationConfig {

  // … as seen before
}
```

If you're using XML configuration, add the `repositories` XML namespace element of the JPA namespace, as shown in Example 4-15.

Example 4-15. Activating JPA repositories through the XML namespace

```
<jpa:repositories base-package="com.acme.repositories" />
```

To see this working, have a look at `CustomerRepositoryIntegrationTest`. It basically uses the Spring configuration set up in `AbstractIntegrationTest`, gets the `CustomerRepository` wired into the test case, and runs the very same tests we find in `JpaCustomerRepositoryIntegrationTest`, only without us having to provide any implementation class for the repository interface whatsoever. Let's look at the individual methods declared in the repository and recap what Spring Data JPA is actually doing for each one of them. See Example 4-16.

Example 4-16. Repository interface definition for Customers

```
public interface CustomerRepository extends Repository<Customer, Long> {

  Customer findOne(Long);

  Customer save(Customer account);

  Customer findByEmailAddress(EmailAddress emailAddress);
}
```

The `findOne(…)` and `save(…)` methods are actually backed by `SimpleJpaRepository`, which is the class of the instance that actually backs the proxy created by the Spring Data infrastructure. So, solely by matching the method signatures, the calls to these two methods get routed to the implementation class. If we wanted to expose a more complete set of CRUD methods, we might simply extend `CrudRepository` instead of `Repository`, as it contains these methods already. Note how we actually prevent `Customer` instances from being deleted by not exposing the `delete(…)` methods that would have been exposed if we had extended `CrudRepository`. Find out more about of the tuning options in "Fine-Tuning Repository Interfaces" on page 20.

The last method to discuss is `findByEmailAddress(…)`, which obviously is not a CRUD one but rather intended to be executed as a query. As we haven't manually declared any, the bootstrapping purpose of Spring Data JPA will inspect the method and try to derive a query from it. The derivation mechanism (details on that in "Query Derivation" on page 17) will discover that `EmailAddress` is a valid property reference for `Customer` and eventually create a JPA Criteria API query whose JPQL equivalent is `select c from Customer c where c.emailAddress = ?1`. Because the method returns a single `Customer`, the query execution expects the query to return at most one resulting entity. If no `Customer` is found, we'll get `null`; if there's more than one found, we'll see a `IncorrectResultSizeDataAccessException`.

Let's continue with the `ProductRepository` interface (Example 4-17). The first thing you note is that compared to `CustomerRepository`, we're extending `CrudRepository` first because we'd like to have the full set of CRUD methods available. The method `findByDescriptionContaining(…)` is clearly a query method. There are several things to note here. First, we not only reference the description property of the product, but also qualify the predicate with the `Containing` keyword. That will eventually lead to the given description parameter being surrounded by `%` characters, and the resulting `String` being

bound via the `LIKE` operator. Thus, the query is as follows: `select p from Product p where p.description like ?1` with a given description of `Apple` bound as `%Apple%`. The second interesting thing is that we're using the pagination abstraction to retrieve only a subset of the products matching the criteria. The `lookupProductsByDescription()` test case in `ProductRepositoryIntegrationTest` shows how that method can be used (Example 4-18).

Example 4-17. Repository interface definition for Products

```
public interface ProductRepository extends CrudRepository<Product, Long> {

  Page<Product> findByDescriptionContaining(String description, Pageable pageable);

  @Query("select p from Product p where p.attributes[?1] = ?2")
  List<Product> findByAttributeAndValue(String attribute, String value);
}
```

Example 4-18. Test case for ProductRepository findByDescriptionContaining(…)

```
@Test
public void lookupProductsByDescription() {

  Pageable pageable = new PageRequest(0, 1, Direction.DESC, "name");
  Page<Product> page = repository.findByDescriptionContaining("Apple", pageable);

  assertThat(page.getContent(), hasSize(1));
  assertThat(page, Matchers.<Product> hasItems(named("iPad")));
  assertThat(page.getTotalElements(), is(2L));
  assertThat(page.isFirstPage(), is(true));
  assertThat(page.isLastPage(), is(false));
  assertThat(page.hasNextPage(), is(true));
}
```

We create a new `PageRequest` instance to ask for the very first page by specifying a page size of one with a descending order by the name of the `Product`. We then simply hand that `Pageable` into the method and make sure we've got the iPad back, that we're the first page, and that there are further pages available. As you can see, the execution of the paging method retrieves the necessary metadata to find out how many items the query would have returned if we hadn't applied pagination. Without Spring Data, reading that metadata would require manually coding the extra query execution, which does a count projection based on the actual query. For more detailed information on pagination with repository methods, see "Pagination and Sorting" on page 18.

The second method in `ProductRepository` is `findByAttributeAndValue(…)`. We'd essentially like to look up all `Products` that have a custom attribute with a given value. Because the attributes are mapped as `@ElementCollection` (see Example 4-5 for reference), we unfortunately cannot use the query derivation mechanism to get the query created for us. To manually define the query to be executed, we use the `@Query` annotation. This also comes in handy if the queries get more complex in general. Even if they were derivable, they'd result in awfully verbose method names.

Finally, let's have a look at the `OrderRepository` (Example 4-19), which should already look remarkably familiar. The query method `findByCustomer(…)` will trigger query derivation (as shown before) and result in `select o from Order o where o.customer = ?1`. The only crucial difference from the other repositories is that we extend `PagingAndSortingRepository`, which in turn extends `CrudRepository`. `PagingAndSortingRepository` adds `findAll(…)` methods that take a `Sort` and `Pageable` parameter on top of what `CrudRepository` already provides. The main use case here is that we'd like to access all `Order`s page by page to avoid loading them all at once.

Example 4-19. Repository interface definition for Orders

```
public interface OrderRepository extends PagingAndSortingRepository<Order, Long> {

  List<Order> findByCustomer(Customer customer);
}
```

Transactionality

Some of the CRUD operations that will be executed against the JPA `EntityManager` require a transaction to be active. To make using Spring Data Repositories for JPA as convenient as possible, the implementation class backing `CrudRepository` and `PagingAndSortingRepository` is equipped with `@Transactional` annotations with a default configuration to let it take part in Spring transactions automatically, or even trigger new ones in case none is already active. For a general introduction into Spring transactions, please consult the Spring reference documentation.

In case the repository implementation actually triggers the transaction, it will create a default one (store-default isolation level, no timeout configured, rollback for runtime exceptions only) for the `save(…)` and `delete(…)` operations and read-only ones for all find methods including the paged ones. Enabling read-only transactions for reading methods results in a few optimizations: first, the flag is handed to the underlying JDBC driver which will—depending on your database vendor—result in optimizations or the driver even preventing you from accidentally executing modifying queries. Beyond that, the Spring transaction infrastructure integrates with the life cycle of the `EntityManager` and can set the `FlushMode` for it to `MANUAL`, preventing it from checking each entity in the persistence context for changes (so-called *dirty checking*). Especially with a large set of objects loaded into the persistence context, this can lead to a significant improvement in performance.

If you'd like to fine-tune the transaction configuration for some of the CRUD methods (e.g., to configure a particular timeout), you can do so by redeclaring the desired CRUD method and adding `@Transactional` with your setup of choice to the method declaration. This will then take precedence over the default configuration declared in `SimpleJpaRepository`. See Example 4-20.

Example 4-20. Reconfiguring transactionality in CustomerRepository interface

```
public interface CustomerRepository extends Repository<Customer, Long> {

  @Transactional(timeout = 60)
  Customer save(Customer account);
}
```

This, of course, also works if you use custom repository base interfaces; see "Fine-Tuning Repository Interfaces" on page 20.

Repository Querydsl Integration

Now that we've seen how to add query methods to repository interfaces, let's look at how we can use Querydsl to dynamically create predicates for entities and execute them via the repository abstraction. Chapter 3 provides a general introduction to what Querydsl actually is and how it works.

To generate the metamodel classes, we have configured the Querydsl Maven plug-in in our *pom.xml* file, as shown in Example 4-21.

Example 4-21. Setting up the Querydsl APT processor for JPA

```
<plugin>
  <groupId>com.mysema.maven</groupId>
  <artifactId>maven-apt-plugin</artifactId>
  <version>1.0.4</version>
  <configuration>
    <processor>com.mysema.query.apt.jpa.JPAAnnotationProcessor</processor>
  </configuration>
  <executions>
    <execution>
      <id>sources</id>
      <phase>generate-sources</phase>
      <goals>
        <goal>process</goal>
      </goals>
      <configuration>
        <outputDirectory>target/generated-sources</outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
```

The only JPA-specific thing to note here is the usage of the `JPAAnnotationProcessor`. It will cause the plug-in to consider JPA mapping annotations to discover entities, relationships to other entities, embeddables, etc. The generation will be run during the normal build process and classes generated into a folder under *target*. Thus, they will be cleaned up with each clean build, and don't get checked into the source control system.

If you're using Eclipse and add the plug-in to your project setup, you will have to trigger a Maven project update (right-click on the project and choose Maven→Update Project...). This will add the configured output directory as an additional source folder so that the code using the generated classes compiles cleanly.

Once this is in place, you should find the generated query classes `QCustomer`, `QProduct`, and so on. Let's explore the capabilities of the generated classes in the context of the `ProductRepository`. To be able to execute Querydsl predicates on the repository, we add the `QueryDslPredicateExecutor` interface to the list of extended types, as shown in Example 4-22.

Example 4-22. The ProductRepository interface extending QueryDslPredicateExecutor

```
public interface ProductRepository extends CrudRepository<Product, Long>,
                                           QueryDslPredicateExecutor<Product> { … }
```

This will pull methods like `findAll(Predicate predicate)` and `findOne(Predicate predicate)` into the API. We now have everything in place, so we can actually start using the generated classes. Let's have a look at the `QuerydslProductRepositoryIntegrationTest` (Example 4-23).

Example 4-23. Using Querydsl predicates to query for Products

```
QProduct product = QProduct.product;

Product iPad = repository.findOne(product.name.eq("iPad"));
Predicate tablets = product.description.contains("tablet");

Iterable<Product> result = repository.findAll(tablets);
assertThat(result, is(Matchers.<Product> iterableWithSize(1)));
assertThat(result, hasItem(iPad));
```

First, we obtain a reference to the `QProduct` metamodel class and keep that inside the product property. We can now use this to navigate the generated path expressions to create predicates. We use a `product.name.eq("iPad")` to query for the `Product` named iPad and keep that as a reference. The second predicate we build specifies that we'd like to look up all products with a description containing `tablet`. We then go on executing the `Predicate` instance against the repository and assert that we found exactly the iPad we looked up for reference before.

You see that the definition of the predicates is remarkably readable and concise. The built predicates can be recombined to construct higher-level predicates and thus allow for querying flexibility without adding complexity.

CHAPTER 5

Type-Safe JDBC Programming with Querydsl SQL

Using JDBC is a popular choice for working with a relational database. Most of Spring's JDBC support is provided in the *spring-jdbc* module of the Spring Framework itself. A good guide for this JDBC support is *Just Spring Data Access* by Madhusudhan Konda [Konda12]. The Spring Data JDBC Extensions subproject of the Spring Data project does, however, provide some additional features that can be quite useful. That's what we will cover in this chapter. We will look at some recent developments around type-safe querying using Querydsl.

In addition to the Querydsl support, the Spring Data JDBC Extensions subproject contains some database-specific support like connection failover, message queuing, and improved stored procedure support for the Oracle database. These features are limited to the Oracle database and are not of general interest, so we won't be covering them in this book. The Spring Data JDBC Extensions subproject does come with a detailed reference guide that covers these features if you are interested in exploring them further.

The Sample Project and Setup

We have been using strings to define database queries in our Java programs for a long time, and as mentioned earlier this can be quite error-prone. Column or table names can change. We might add a column or change the type of an existing one. We are used to doing similar refactoring for our Java classes in our Java IDEs, and the IDE will guide us so we can find any references that need changing, including in comments and configuration files. No such support is available for strings containing complex SQL query expressions. To avoid this problem, we provide support for a type-safe query alternative in Querydsl. Many data access technologies integrate well with Querydsl, and Chapter 3 provided some background on it. In this section we will focus on the Querydsl SQL module and how it integrates with Spring's `JdbcTemplate` usage, which should be familiar to every Spring developer.

Before we look at the new JDBC support, however, we need to discuss some general concerns like database configuration and project build system setup.

The HyperSQL Database

We are using the HyperSQL database version 2.2.8 for our Querydsl examples in this chapter. One nice feature of HyperSQL is that we can run the database in both server mode and in-memory. The in-memory option is great for integration tests since starting and stopping the database can be controlled by the application configuration using Spring's `EmbeddedDatabaseBuilder`, or the `<jdbc:embedded-database>` tag when using the spring-jdbc XML namespace. The build scripts download the dependency and start the in-memory database automatically. To use the database in standalone server mode, we need to download the distribution and unzip it to a directory on our system. Once that is done, we can change to the *hsqldb* directory of the unzipped distribution and start the database using this command:

```
java -classpath lib/hsqldb.jar org.hsqldb.server.Server \
  --database.0 file:data/test --dbname.0 test
```

Running this command starts up the server, which generates some log output and a message that the server has started. We are also told we can use Ctrl-C to stop the server. We can now open another command window, and from the same *hsqldb* directory we can start up a database client so we can interact with the database (creating tables and running queries, etc.). For Windows, we need to execute only the *runManagerSwing.bat* batch file located in the *bin* directory. For OS X or Linux, we can run the following command:

```
java -classpath lib/hsqldb.jar org.hsqldb.util.DatabaseManagerSwing
```

This should bring up the login dialog shown in Figure 5-1. We need to change the Type to HSQL Database Engine Server and add "test" as the name of the database to the URL so it reads `jdbc:hsqldb:hsql://localhost/test`. The default user is "sa" with a blank password. Once connected, we have an active GUI database client.

The SQL Module of Querydsl

The SQL module of Querydsl provides a type-safe option for the Java developer to work with relational databases. Instead of writing SQL queries and embedding them in strings in your Java program, Querydsl generates query types based on metadata from your database tables. You use these generated types to write your queries and perform CRUD operations against the database without having to resort to providing column or table names using strings.

The way you generate the query types is a bit different in the SQL module compared to other Querydsl modules. Instead of relying on annotations, the SQL module relies on the actual database tables and available JDBC metadata for generating the query

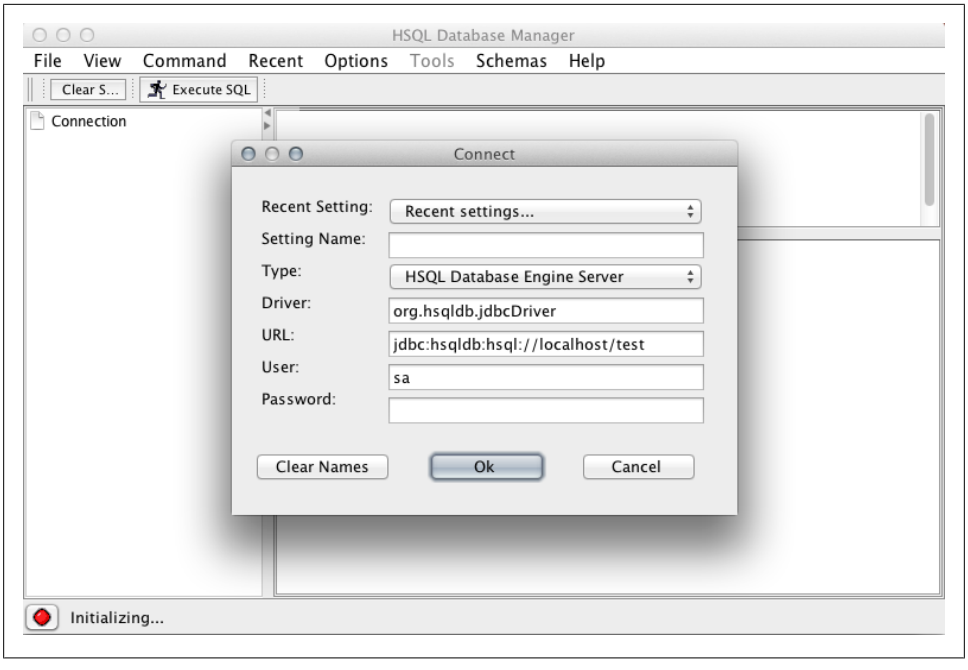


Figure 5-1. HSQLDB client login dialog

types. This means that you need to have the tables created and access to a live database before you run the query class generation. For this reason, we recommend running this as a separate step of the build and saving the generated classes as part of the project in the source control system. We need to rerun this step only when we have made some changes to our table structures and before we check in our code. We expect the continuous integration system to run this code generation step as well, so any mismatch between the Java types and the database tables would be detected at build time.

We'll take a look at what we need to generate the query types later, but first we need to understand what they contain and how we use them. They contain information that Querydsl can use to generate queries, but they also contain information you can use to compose queries; perform updates, inserts, and deletes; and map data to domain objects. Let's take a quick look at an example of a table to hold address information. The `address` table has three `VARCHAR` columns: `street`, `city`, and `country`. Example 5-1 shows the SQL statement to create this table.

Example 5-1. Creating the address table

```
CREATE TABLE address (
  id BIGINT IDENTITY PRIMARY KEY,
  customer_id BIGINT CONSTRAINT address_customer_ref
    FOREIGN KEY REFERENCES customer (id),
  street VARCHAR(255),
  city VARCHAR(255),
  country VARCHAR(255));
```

Example 5-2 demonstrates the generated query type based on this `address` table. It has some constructors, Querydsl path expressions for the columns, methods to create primary and foreign key types, and a static field that provides an instance of the `QAddress` class.

Example 5-2. A generated query type—QAddress

```
package com.oreilly.springdata.jdbc.domain;

import static com.mysema.query.types.PathMetadataFactory.*;

import com.mysema.query.types.*;
import com.mysema.query.types.path.*;

import javax.annotation.Generated;


/**
 * QAddress is a Querydsl query type for QAddress
 */
@Generated("com.mysema.query.sql.codegen.MetaDataSerializer")
public class QAddress extends com.mysema.query.sql.RelationalPathBase<QAddress> {

  private static final long serialVersionUID = 207732776;

  public static final QAddress address = new QAddress("ADDRESS");

  public final StringPath city = createString("CITY");
  public final StringPath country = createString("COUNTRY");
  public final NumberPath<Long> customerId = createNumber("CUSTOMER_ID", Long.class);
  public final NumberPath<Long> id = createNumber("ID", Long.class);
  public final StringPath street = createString("STREET");
  public final com.mysema.query.sql.PrimaryKey<QAddress> sysPk10055 = createPrimaryKey(id);
  public final com.mysema.query.sql.ForeignKey<QCustomer> addressCustomerRef =
      createForeignKey(customerId, "ID");

  public QAddress(String variable) {
    super(QAddress.class, forVariable(variable), "PUBLIC", "ADDRESS");
  }

  public QAddress(Path<? extends QAddress> entity) {
    super(entity.getType(), entity.getMetadata(), "PUBLIC", "ADDRESS");
  }

  public QAddress(PathMetadata<?> metadata) {
    super(QAddress.class, metadata, "PUBLIC", "ADDRESS");
  }
}
```

By creating a reference like this:

```
QAddress qAddress = QAddress.address;
```

in our Java code, we can reference the table and the columns more easily using `qAddress` instead of resorting to using string literals.

In Example 5-3, we query for the street, city, and country for any address that has London as the city.

Example 5-3. Using the generated query class

```
QAddress qAddress = QAddress.address;
SQLTemplates dialect = new HSQLDBTemplates();
SQLQuery query = new SQLQueryImpl(connection, dialect)
    .from(qAddress)
    .where(qAddress.city.eq("London"));
List<Address> results = query.list(
    new QBean<Address>(Address.class, qAddress.street,
      qAddress.city, qAddress.country));
```

First, we create a reference to the query type and an instance of the correct `SQLTemplates` for the database we are using, which in our case is `HSQLDBTemplates`. The `SQLTemplates` encapsulate the differences between databases and are similar to Hibernate's `Dialect`. Next, we create an `SQLQuery` with the JDBC `javax.sql.Connection` and the `SQLTemplates` as the parameters. We specify the table we are querying using the `from` method, passing in the query type. Next, we provide the where clause or predicate via the `where` method, using the `qAddress` reference to specify the criteria that `city` should equal `London`.

Executing the `SQLQuery`, we use the `list` method, which will return a `List` of results. We also provide a mapping implementation using a `QBean`, parameterized with the domain type and a projection consisting of the columns `street`, `city`, and `country`.

The result we get back is a `List` of `Addresses`, populated by the `QBean`. The `QBean` is similar to Spring's `BeanPropertyRowMapper`, and it requires that the domain type follows the `JavaBean` style. Alternatively, you can use a `MappingProjection`, which is similar to Spring's familiar `RowMapper` in that you have more control over how the results are mapped to the domain object.

Based on this brief example, let's summarize the components of Querydsl that we used for our SQL query:

- The `SQLQueryImpl` class , which will hold the target table or tables along with the predicate or where clause and possibly a join expression if we are querying multiple tables
- The `Predicate`, usually in the form of a `BooleanExpression` that lets us specify filters on the results
- The mapping or results extractor, usually in the form of a `QBean` or `MappingProjection` parameterized with one or more `Expressions` as the projection

So far, we haven't integrated with any Spring features, but the rest of the chapter covers this integration. This first example is just intended to introduce the basics of the Querydsl SQL module.

Build System Integration

The code for the Querydsl part of this chapter is located in the *jdbc* module of the sample GitHub project.

Before we can really start using Querydsl in our project, we need to configure our build system so that we can generate the query types. Querydsl provides both Maven and Ant integration, documented in the "Querying SQL" chapter of the Querydsl reference documentation.

In our Maven *pom.xml* file, we add the plug-in configuration shown in Example 5-4.

Example 5-4. Setting up code generation Maven plug-in

```
<plugin>
  <groupId>com.mysema.querydsl</groupId>
  <artifactId>querydsl-maven-plugin</artifactId>
  <version>${querydsl.version}</version>
  <configuration>
    <jdbcDriver>org.hsqldb.jdbc.JDBCDriver</jdbcDriver>
    <jdbcUrl>jdbc:hsqldb:hsql://localhost:9001/test</jdbcUrl>
    <jdbcUser>sa</jdbcUser>
    <schemaPattern>PUBLIC</schemaPattern>
    <packageName>com.oreilly.springdata.jdbc.domain</packageName>
    <targetFolder>${project.basedir}/src/generated/java</targetFolder>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>org.hsqldb</groupId>
      <artifactId>hsqldb</artifactId>
      <version>2.2.8</version>
    </dependency>
    <dependency>
      <groupId>ch.qos.logback</groupId>
      <artifactId>logback-classic</artifactId>
      <version>${logback.version}</version>
    </dependency>
  </dependencies>
</plugin>
```

We will have to execute this plug-in explicitly using the following Maven command:

```
mvn com.mysema.querydsl:querydsl-maven-plugin:export
```

You can set the plug-in to execute as part of the `generate-sources` life cycle phase by specifying an execution goal. We actually do this in the example project, and we also use a predefined HSQL database just to avoid forcing you to start up a live database when you build the example project. For real work, though, you do need to have a database where you can modify the schema and rerun the Querydsl code generation.

The Database Schema

Now that we have the build configured, we can generate the query classes, but let's first review the database schema that we will be using for this section. We already saw the `address` table, and we are now adding a `customer` table that has a one-to-many relationship with the `address` table. We define the schema for our HSQLDB database as shown in Example 5-5.

Example 5-5. schema.sql

```
CREATE TABLE customer (
  id BIGINT IDENTITY PRIMARY KEY,
  first_name VARCHAR(255),
  last_name VARCHAR(255),
  email_address VARCHAR(255));

CREATE UNIQUE INDEX ix_customer_email ON CUSTOMER (email_address ASC);

CREATE TABLE address (
  id BIGINT IDENTITY PRIMARY KEY,
  customer_id BIGINT CONSTRAINT address_customer_ref FOREIGN KEY REFERENCES customer (id),
  street VARCHAR(255),
  city VARCHAR(255),
  country VARCHAR(255));
```

The two tables, `customer` and `address`, are linked by a foreign key reference from `address` to `customer`. We also define a unique index on the `email_address` column of the `address` table.

This gives us the domain model implementation shown in Figure 5-2.

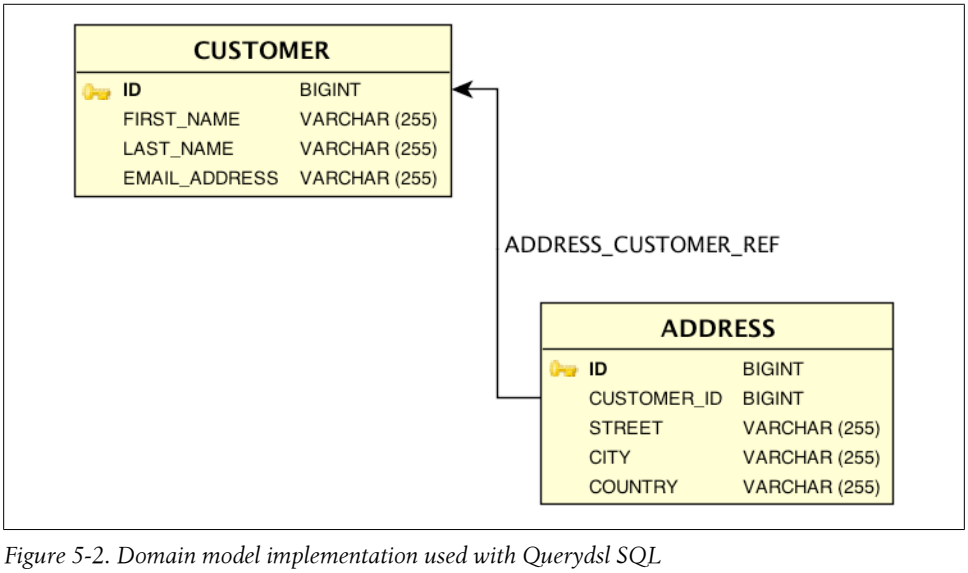


Figure 5-2. Domain model implementation used with Querydsl SQL

The Domain Implementation of the Sample Project

We have already seen the schema for the database, and now we will take a look at the corresponding Java domain classes we will be using for our examples. We need a `Customer` class plus an `Address` class to hold the data from our database tables. Both of these classes extend an `AbstractEntity` class that, in addition to `equals(…)` and `hashCode()`, has setters and getters for the `id`, which is a `Long`:

```
public class AbstractEntity {

  private Long id;

  public Long getId() {
    return id;
  }

  public void setId(Long id) {
    this.id = id;
  }

  @Override
  public boolean equals(Object obj) { … }

  @Override
  public int hashCode() { … }
}
```

The `Customer` class has name and email information along with a set of addresses. This implementation is a traditional JavaBean with getters and setters for all properties:

```
public class Customer extends AbstractEntity {

  private String firstName;
  private String lastName;
  private EmailAddress emailAddress;
  private Set<Address> addresses = new HashSet<Address>();

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public EmailAddress getEmailAddress() {
    return emailAddress;
```

```
  }

  public void setEmailAddress(EmailAddress emailAddress) {
    this.emailAddress = emailAddress;
  }

  public Set<Address> getAddresses() {
    return Collections.unmodifiableSet(addresses);
  }

  public void addAddress(Address address) {
    this.addresses.add(address);
  }

  public void clearAddresses() {
    this.addresses.clear();
  }
}
```

The email address is stored as a `VARCHAR` column in the database, but in the Java class we use an `EmailAddress` value object type that also provides validation of the email address using a regular expression. This is the same class that we have seen in the other chapters:

```
public class EmailAddress {

  private static final String EMAIL_REGEX = …;
  private static final Pattern PATTERN = Pattern.compile(EMAIL_REGEX);

  private String value;

  protected EmailAddress() {
  }

  public EmailAddress(String emailAddress) {
    Assert.isTrue(isValid(emailAddress), "Invalid email address!");
    this.value = emailAddress;
  }

  public static boolean isValid(String source) {
    return PATTERN.matcher(source).matches();
  }
}
```

The last domain class is the `Address` class, again a traditional JavaBean with setters and getters for the address properties. In addition to the no-argument constructor, we have a constructor that takes all address properties:

```
public class Address extends AbstractEntity {
  private String street, city, country;

  public Address() {
  }
```

```
  public Address(String street, String city, String country) {
    this.street = street;
    this.city = city;
    this.country = country;
  }

  public String getCountry() {
    return country;
  }

  public void setCountry(String country) {
    this.country = country;
  }

  public String getStreet() {
    return street;
  }

  public void setStreet(String street) {
    this.street = street;
  }

  public String getCity() {
    return city;
  }

  public void setCity(String city) {
    this.city = city;
  }
}
```

The preceding three classes make up our domain model and reside in the `com.oreilly.springdata.jdbc.domain` package of the JDBC example project. Now we are ready to look at the interface definition of our `CustomerRepository`:

```
public interface CustomerRepository {

  Customer findById(Long id);

  List<Customer> findAll();

  void save(Customer customer);

  void delete(Customer customer);

  Customer findByEmailAddress(EmailAddress emailAddress);
}
```

We have a couple of finder methods and save and delete methods. We don't have any repository methods to save and delete the `Address` objects since they are always owned by the `Customer` instances. We will have to persist any addresses provided when the `Customer` instance is saved.

The QueryDslJdbcTemplate

The central class in the Spring Data integration with Querydsl is the `QueryDslJdbcTemplate`. It is a wrapper around a standard Spring `JdbcTemplate` that has methods for managing `SQLQuery` instances and executing queries as well as methods for executing inserts, updates, and deletes using command-specific callbacks. We'll cover all of these in this section, but let's start by creating a `QueryDslJdbcTemplate`.

To configure the `QueryDslJdbcTemplate`, you simply pass in either a `DataSource`:

```
QueryDslJdbcTemplate qdslTemplate = new QueryDslJdbcTemplate(dataSource);
```

or an already configured `JdbcTemplate` in the constructor:

```
jdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
QueryDslJdbcTemplate qdslTemplate = new QueryDslJdbcTemplate(jdbcTemplate);
```

Now we have a fully configured `QueryDslJdbcTemplate` to use. We saw earlier that usually you need to provide a `Connection` and an `SQLTemplates` matching your database when you create an `SQLQuery` object. However, when you use the `QueryDslJdbcTemplate`, there is no need to do this. In usual Spring fashion, the JDBC layer will manage any database resources like connections and result sets. It will also take care of providing the `SQLTemplates` instance based on database metadata from the managed connection. To obtain a managed instance of an `SQLQuery`, you use the `newSqlQuery` static factory method of the `QueryDslJdbcTemplate`:

```
SQLQuery sqlQuery = qdslTemplate.newSqlQuery();
```

The `SQLQuery` instance obtained does not yet have a live connection, so you need to use the query methods of the `QueryDslJdbcTemplate` to allow connection management to take place:

```
SQLQuery addressQuery = qdslTemplate.newSqlQuery()
  .from(qAddress)
  .where(qAddress.city.eq("London"));

List<Address> results = qdslTemplate.query(
  addressQuery,
  BeanPropertyRowMapper.newInstance(Address.class),
  qAddress.street, qAddress.city, qAddress.country);
```

There are two query methods: `query` returning a `List` and `queryForObject` returning a single result. Both of these have three overloaded versions, each taking the following parameters:

- `SQLQuery` object obtained via the `newSqlQuery` factory method
- One of the following combinations of a mapper and projection implementation:
  - `RowMapper`, plus a projection expressed as one or more `Expression`s
  - `ResultSetExtractor`, plus a projection expressed as one or more `Expression`s
  - `ExpressionBase`, usually expressed as a `QBean` or `MappingProjection`

The first two mappers, `RowMapper` and `ResultSetExtractor`, are standard Spring interfaces often used with the regular `JdbcTemplate`. They are responsible for extracting the data from the results returned by a query. The `ResultSetExtractor` extracts data for all rows returned, while the `RowMapper` handles only one row at the time and will be called repeatedly, once for each row. `QBean` and `MappingProjection` are Querydsl classes that also map one row at the time. Which ones you use is entirely up to you; they all work equally well. For most of our examples, we will be using the Spring types—this book is called *Spring Data*, after all.

Executing Queries

Now we will look at how we can use the `QueryDslJdbcTemplate` to execute queries by examining how we should implement the query methods of our `CustomerRepository`.

The Beginning of the Repository Implementation

The implementation will be autowired with a `DataSource`; in that setter, we will create a `QueryDslJdbcTemplate` and a projection for the table columns used by all queries when retrieving data needed for the `Customer` instances. (See Example 5-6.)

Example 5-6. Setting up the QueryDslCustomerRepository instance

```
@Repository
@Transactional
public class QueryDslCustomerRepository implements CustomerRepository {

  private final QCustomer qCustomer = QCustomer.customer;
  private final QAddress qAddress = QAddress.address;

  private final QueryDslJdbcTemplate template;
  private final Path<?>[] customerAddressProjection;

  @Autowired
  public QueryDslCustomerRepository(DataSource dataSource) {

    Assert.notNull(dataSource);

    this.template = new QueryDslJdbcTemplate(dataSource);
    this.customerAddressProjection = new Path<?>[] { qCustomer.id, qCustomer.firstName,
      qCustomer.lastName, qCustomer.emailAddress, qAddress.id, qAddress.customerId,
      qAddress.street, qAddress.city, qAddress.country };
  }

  @Override
  @Transactional(readOnly = true)
  public Customer findById(Long id) { … }

  @Override
  @Transactional(readOnly = true)
  public Customer findByEmailAddress(EmailAddress emailAddress) { … }
```

```
  @Override
  public void save(final Customer customer) { … }

  @Override
  public void delete(final Customer customer) { … }
}
```

We are writing a repository, so we start off with an `@Repository` annotation. This is a standard Spring stereotype annotation, and it will make your component discoverable during classpath scanning. In addition, for repositories that use ORM-style data access technologies, it will also make your repository a candidate for exception translation between the ORM-specific exceptions and Spring's `DataAccessException` hierarchy. In our case, we are using a template-based approach, and the template itself will provide this exception translation.

Next is the `@Transactional` annotation. This is also a standard Spring annotation that indicates that any call to a method in this class should be wrapped in a database transaction. As long as we provide a transaction manager implementation as part of our configuration, we don't need to worry about starting and completing these transactions in our repository code.

We also define two references to the two query types that we have generated, `QCustomer` and `QAddress`. The array `customerAddressProjection` will hold the Querydsl `Path` entries for our queries, one `Path` for each column we are retrieving.

The constructor is annotated with `@Autowired`, which means that when the repository implementation is configured, the Spring container will inject the `DataSource` that has been defined in the application context. The rest of the class comprises the methods from the `CustomerRepository` that we need to provide implementations for, so let's get started.

Querying for a Single Object

First, we will implement the `findById` method (Example 5-7). The ID we are looking for is passed in as the only argument. Since this is a read-only method, we can add a `@Transactional(readOnly = true)` annotation to provide a hint that some JDBC drivers will use to improve transaction handling. It never hurts to provide this optional attribute for read-only methods even if some JDBC drivers won't make use of it.

Example 5-7. Query for single object

```
@Transactional(readOnly = true)
public Customer findById(Long id) {

  SQLQuery findByIdQuery = template.newSqlQuery()
    .from(qCustomer)
    .leftJoin(qCustomer._addressCustomerRef, qAddress)
    .where(qCustomer.id.eq(id));
```

```
  return template.queryForObject(
    findByIdQuery,
    new CustomerExtractor(),
    customerAddressProjection);
}
```

We start by creating an `SQLQuery` instance. We have already mentioned that when we are using the `QueryDslJdbcTemplate`, we need to let the template manage the `SQLQuery` instances. That's why we use the factory method `newSqlQuery()` to obtain an instance. The `SQLQuery` class provides a fluent interface where the methods return the instance of the `SQLQuery`. This makes it possible to string a number of methods together, which in turn makes it easier to read the code. We specify the main table we are querying (the `customer` table) with the `from` method. Then we add a left outer join against the `address` table using the `leftJoin(…)` method. This will include any address rows that match the foreign key reference between address and customer. If there are none, the address columns will be `null` in the returned results. If there is more than one address, we will get multiple rows for each customer, one for each address row. This is something we will have to handle in our mapping to the Java objects later on. The last part of the `SQLQuery` is specifying the predicate using the `where` method and providing the criteria that the `id` column should equal the `id` parameter.

After we have the `SQLQuery` created, we execute our query by calling the `queryForObject` method of the `QueryDslJdbcTemplate`, passing in the `SQLQuery` and a combination of a mapper and a projection. In our case, that is a `ResultSetExtractor` and the `customerAddressProjection` that we created earlier. Remember that we mentioned earlier that since our query contained a `leftJoin`, we needed to handle potential multiple rows per `Customer`.

Example 5-8 is the implementation of this `CustomerExtractor`.

Example 5-8. CustomerExtractor for single object

```
private static class CustomerExtractor implements ResultSetExtractor<Customer> {

  CustomerListExtractor customerListExtractor =
    new CustomerListExtractor(OneToManyResultSetExtractor.ExpectedResults.ONE_OR_NONE);

  @Override
  public Customer extractData(ResultSet rs) throws SQLException, DataAccessException {

    List<Customer> list = customerListExtractor.extractData(rs);
    return list.size() > 0 ? list.get(0) : null;
  }
}
```

As you can see, we use a `CustomerListExtractor` that extracts a `List` of `Customer` objects, and we return the first object in the `List` if there is one, or `null` if the `List` is empty. We know that there could not be more than one result since we set the parameter `expectedResults` to `OneToManyResultSetExtractor.ExpectedResults.ONE_OR_NONE` in the constructor of the `CustomerListExtractor`.

The OneToManyResultSetExtractor Abstract Class

Before we look at the `CustomerListExtractor`, let's look at the base class, which is a special implementation named `OneToManyResultSetExtractor` that is provided by the Spring Data JDBC Extension project. Example 5-9 gives an outline of what the `OneToManyResultSetExtractor` provides.

Example 5-9. Outline of OneToManyResultSetExtractor for extracting List of objects

```
public abstract class OneToManyResultSetExtractor<R, C, K>
    implements ResultSetExtractor<List<R>> {

  public enum ExpectedResults {
    ANY,
    ONE_AND_ONLY_ONE,
    ONE_OR_NONE,
    AT_LEAST_ONE
  }

  protected final ExpectedResults expectedResults;
  protected final RowMapper<R> rootMapper;
  protected final RowMapper<C> childMapper;

  protected List<R> results;

  public OneToManyResultSetExtractor(RowMapper<R> rootMapper, RowMapper<C> childMapper) {
    this(rootMapper, childMapper, null);
  }

  public OneToManyResultSetExtractor(RowMapper<R> rootMapper, RowMapper<C> childMapper,
      ExpectedResults expectedResults) {

    Assert.notNull(rootMapper);
    Assert.notNull(childMapper);

    this.rootMapper = rootMapper;
    this.childMapper = childMapper;
    this.expectedResults = expectedResults == null ? ExpectedResults.ANY : expectedResults;
  }

  public List<R> extractData(ResultSet rs) throws SQLException, DataAccessException { … }

  /**
   * Map the primary key value to the required type.
   * This method must be implemented by subclasses.
   * This method should not call {@link ResultSet#next()}
   * It is only supposed to map values of the current row.
   *
   * @param rs the ResultSet
   * @return the primary key value
   * @throws SQLException
   */
  protected abstract K mapPrimaryKey(ResultSet rs) throws SQLException;
```

```
  /**
   * Map the foreign key value to the required type.
   * This method must be implemented by subclasses.
   * This method should not call {@link ResultSet#next()}.
   * It is only supposed to map values of the current row.
   *
   * @param rs the ResultSet
   * @return the foreign key value
   * @throws SQLException
   */
  protected abstract K mapForeignKey(ResultSet rs) throws SQLException;

  /**
   * Add the child object to the root object
   * This method must be implemented by subclasses.
   *
   * @param root the Root object
   * @param child the Child object
   */
  protected abstract void addChild(R root, C child);
}
```

This `OneToManyResultSetExtractor` extends the `ResultSetExtractor`, parameterized with `List<T>` as the return type. The method `extractData` is responsible for iterating over the `ResultSet` and extracting row data. The `OneToManyResultSetExtractor` has three abstract methods that must be implemented by subclasses `mapPrimaryKey`, `mapForeignKey`, and `addChild`. These methods are used when iterating over the result set to identify both the primary key and the foreign key so we can determine when there is a new root, and to help add the mapped child objects to the root object.

The `OneToManyResultSetExtractor` class also needs `RowMapper` implementations to provide the mapping required for the root and child objects.

The CustomerListExtractor Implementation

Now, let's move on and look at the actual implementation of the `CustomerListExtractor` responsible for extracting the results of our customer and address results. See Example 5-10.

Example 5-10. CustomerListExtractor implementation for extracting List of objects

```
private static class CustomerListExtractor
  extends OneToManyResultSetExtractor<Customer, Address, Integer> {

  private static final QCustomer qCustomer = QCustomer.customer;

  private final QAddress qAddress = QAddress.address;

  public CustomerListExtractor() {
    super(new CustomerMapper(), new AddressMapper());
  }
```

```
  public CustomerListExtractor(ExpectedResults expectedResults) {
    super(new CustomerMapper(), new AddressMapper(), expectedResults);
  }

  @Override
  protected Integer mapPrimaryKey(ResultSet rs) throws SQLException {
    return rs.getInt(qCustomer.id.toString());
  }

  @Override
  protected Integer mapForeignKey(ResultSet rs) throws SQLException {
    String columnName = qAddress.addressCustomerRef.getLocalColumns().get(0).toString();
    if (rs.getObject(columnName) != null) {
      return rs.getInt(columnName);
    } else {
      return null;
    }
  }

  @Override
  protected void addChild(Customer root, Address child) {
    root.addAddress(child);
  }
}
```

The `CustomerListExtractor` extends this `OneToManyResultSetExtractor`, calling the superconstructor passing in the needed mappers for the `Customer` class, `CustomerMapper` (which is the root of the one-to-many relationship), and the mapper for the `Address` class, `AddressMapper` (which is the child of the same one-to-many relationship).

In addition to these two mappers, we need to provide implementations for the `mapPrimaryKey`, `mapForeignKey`, and `addChild` methods of the abstract `OneToManyResultSetExtractor` class.

Next, we will take a look at the `RowMapper` implementations we are using.

The Implementations for the RowMappers

The `RowMapper` implementations we are using are just what you would use with the regular `JdbcTemplate`. They implement a method named `mapRow` with a `ResultSet` and the row number as parameters. The only difference with using a `QueryDslJdbcTemplate` is that instead of accessing the columns with string literals, you use the query types to reference the column labels. In the `CustomerRepository`, we provide a static method for extracting this label via the `toString` method of the `Path`:

```
  private static String columnLabel(Path<?> path) {
    return path.toString();
  }
```

Since we implement the `RowMapper`s as static inner classes, they have access to this private static method.

First, let's look at the mapper for the `Customer` object. As you can see in Example 5-11, we reference columns specified in the `qCustomer` reference to the `QCustomer` query type.

Example 5-11. Root RowMapper implementation for Customer

```
private static class CustomerMapper implements RowMapper<Customer> {

  private static final QCustomer qCustomer = QCustomer.customer;

  @Override
  public Customer mapRow(ResultSet rs, int rowNum) throws SQLException {

    Customer c = new Customer();

    c.setId(rs.getLong(columnLabel(qCustomer.id)));
    c.setFirstName(rs.getString(columnLabel(qCustomer.firstName)));
    c.setLastName(rs.getString(columnLabel(qCustomer.lastName)));

    if (rs.getString(columnLabel(qCustomer.emailAddress)) != null) {
      c.setEmailAddress(
          new EmailAddress(rs.getString(columnLabel(qCustomer.emailAddress))));
    }

    return c;
  }
}
```

Next, we look at the mapper for the `Address` objects, using a `qAddress` reference to the `QAddress` query type (Example 5-12).

Example 5-12. Child RowMapper implementation for Address

```
private static class AddressMapper implements RowMapper<Address> {

  private final QAddress qAddress = QAddress.address;

  @Override
  public Address mapRow(ResultSet rs, int rowNum) throws SQLException {

    String street = rs.getString(columnLabel(qAddress.street));
    String city = rs.getString(columnLabel(qAddress.city));
    String country = rs.getString(columnLabel(qAddress.country));

    Address a = new Address(street, city, country);
    a.setId(rs.getLong(columnLabel(qAddress.id)));

    return a;
  }
}
```

Since the `Address` class has setters for all properties, we could have used a standard Spring `BeanPropertyRowMapper` instead of providing a custom implementation.

Querying for a List of Objects

When it comes to querying for a list of objects, the process is exactly the same as for querying for a single object except that you now can use the `CustomerListExtractor` directly without having to wrap it and get the first object of the `List`. See Example 5-13.

Example 5-13. Query for list of objects

```
@Transactional(readOnly = true)
public List<Customer> findAll() {

  SQLQuery allCustomersQuery = template.newSqlQuery()
      .from(qCustomer)
      .leftJoin(qCustomer._addressCustomerRef, qAddress);

  return template.query(
      allCustomersQuery,
      new CustomerListExtractor(),
      customerAddressProjection);
}
```

We create an `SQLQuery` using the `from(…)` and `leftJoin(…)` methods, but this time we don't provide a predicate since we want all customers returned. When we execute this query, we use the `CustomerListExtractor` directly and the same `customerAddressProjection` that we used earlier.

Insert, Update, and Delete Operations

We will finish the `CustomerRepository` implementation by adding some insert, update, and delete capabilities in addition to the query features we just discussed. With Querydsl, data is manipulated via operation-specific clauses like `SQLInsertClause`, `SQLUpdateClause`, and `SQLDeleteClause`. We will cover how to use them with the `QueryDslJdbcTemplate` in this section.

Inserting with the SQLInsertClause

When you want to insert some data into the database, Querydsl provides the `SQLInsertClause` class. Depending on whether your tables autogenerate the key or you provide the key explicitly, there are two different `execute(…)` methods. For the case where the keys are autogenerated, you would use the `executeWithKey(…)` method. This method will return the generated key so you can set that on your domain object. When you provide the key, you instead use the `execute` method, which returns the number of affected rows. The `QueryDslJdbcTemplate` has two corresponding methods: `insertWithKey(…)` and `insert(…)`.

We are using autogenerated keys, so we will be using the `insertWithKey(…)` method for our inserts, as shown in Example 5-14. The `insertWithKey(…)` method takes a reference to the query type and a callback of type `SqlInsertWithKeyCallback` parameterized with

the type of the generated key. The `SqlInsertWithKeyCallback` callback interface has a single method named `doInSqlInsertWithKeyClause(…)`. This method has the `SQLInsertClause` as its parameter. We need to set the values using this `SQLInsertClause` and then call `executeWithKey(…)`. The key that gets returned from this call is the return value of the `doInSqlInsertWithKeyClause`.

Example 5-14. Inserting an object

```
Long generatedKey = qdslTemplate.insertWithKey(qCustomer,
  new SqlInsertWithKeyCallback<Long>() {

    @Override
    public Long doInSqlInsertWithKeyClause(SQLInsertClause insert) throws SQLException {

      EmailAddress emailAddress = customer.getEmailAddress();
      String emailAddressString = emailAddress == null ? null : emailAddress.toString();

      return insert.columns(
              qCustomer.firstName, qCustomer.lastName, qCustomer.emailAddress)
          .values(customer.getFirstName(), customer.getLastName(), emailAddress);
          .executeWithKey(qCustomer.id);
    }
  });

customer.setId(generatedKey);
```

Updating with the SQLUpdateClause

Performing an update operation is very similar to the insert except that we don't have to worry about generated keys. The method on the `QueryDslJdbcTemplate` is called `update`, and it expects a reference to the query type and a callback of type `SqlUpdateCallback`. The `SqlUpdateCallback` has the single method `doInSqlUpdateClause(…)` with the `SQLUpdateClause` as the only parameter. After setting the values for the update and specifying the where clause, we call `execute` on the `SQLUpdateClause`, which returns an update count. This update count is also the value we need to return from this callback. See Example 5-15.

Example 5-15. Updating an object

```
qdslTemplate.update(qCustomer, new SqlUpdateCallback() {

  @Override
  public long doInSqlUpdateClause(SQLUpdateClause update) {

    EmailAddress emailAddress = customer.getEmailAddress();
    String emailAddressString = emailAddress == null ? null : emailAddress.toString();

    return update.where(qCustomer.id.eq(customer.getId()))
        .set(qCustomer.firstName, customer.getFirstName())
        .set(qCustomer.lastName, customer.getLastName())
        .set(qCustomer.emailAddress, emailAddressString)
        .execute();
```

```
  }
});
```

Deleting Rows with the SQLDeleteClause

Deleting is even simpler than updating. The `QueryDslJdbcTemplate` method you use is called `delete`, and it expects a reference to the query type and a callback of type `SqlDeleteCallback`. The `SqlDeleteCallback` has the single method `doInSqlDeleteClause` with the `SQLDeleteClause` as the only parameter. There's no need to set any values here—just provide the where clause and call execute. See Example 5-16.

Example 5-16. Deleting an object

```
qdslTemplate.delete(qCustomer, new SqlDeleteCallback() {

  @Override
  public long doInSqlDeleteClause(SQLDeleteClause delete) {
    return delete.where(qCustomer.id.eq(customer.getId())).execute();
  }
});
```

PART III

NoSQL

CHAPTER 6

MongoDB: A Document Store

This chapter will introduce you to the Spring Data MongoDB project. We will take a brief look at MongoDB as a document store and explain you how to set it up and configure it to be usable with our sample project. A general overview of MongoDB concepts and the native Java driver API will round off the introduction. After that, we'll discuss the Spring Data MongoDB module's features, the Spring namespace, how we model the domain and map it to the store, and how to read and write data using the `MongoTemplate`, the core store interaction API. The chapter will conclude by discussing the implementation of a data access layer for our domain using the Spring Data repository abstraction.

MongoDB in a Nutshell

MongoDB is a document data store. Documents are structured data—basically maps—that can have primitive values, collection values, or even nested documents as values for a given key. MongoDB stores these documents in BSON, a binary derivative of JSON. Thus, a sample document would look something like Example 6-1.

Example 6-1. A sample MongoDB document

```
{ firstname : "Dave",
  lastname : "Matthews",
  addresses : [ { city : "New York", street : "Broadway" } ] }
```

As you can see, we have primitive `String` values for `firstname` and `lastname`. The `addresses` field has an array value that in turn contains a nested address document. Documents are organized in *collections*, which are arbitrary containers for a set of documents. Usually, you will keep documents of the same type inside a single collection, where *type* essentially means "similarly structured." From a Java point of view, this usually reads as a collection per type (one for `Customer`s, one for `Product`s) or type hierarchy (a single collection to hold `Contact`s, which can either be `Person`s or `Companie`s).

Setting Up MongoDB

To start working with MongoDB, you need to download it from the project's website. It provides binaries for Windows, OS X, Linux, and Solaris, as well as the sources. The easiest way to get started is to just grab the binaries and unzip them to a reasonable folder on your hard disk, as shown in Example 6-2.

Example 6-2. Downloading and unzipping MongoDB distribution

```
$ cd ~/dev
$ curl http://fastdl.mongodb.org/osx/mongodb-osx-x86_64-2.0.6.tgz > mongo.tgz

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 41.1M  100 41.1M    0     0   704k      0  0:00:59  0:00:59 --:--:--  667k

$ tar -zxvf mongo.tgz

x mongodb-osx-x86_64-2.0.6/
x mongodb-osx-x86_64-2.0.6/bin/
x mongodb-osx-x86_64-2.0.6/bin/bsondump
x mongodb-osx-x86_64-2.0.6/bin/mongo
x mongodb-osx-x86_64-2.0.6/bin/mongod
x mongodb-osx-x86_64-2.0.6/bin/mongodump
x mongodb-osx-x86_64-2.0.6/bin/mongoexport
x mongodb-osx-x86_64-2.0.6/bin/mongofiles
x mongodb-osx-x86_64-2.0.6/bin/mongoimport
x mongodb-osx-x86_64-2.0.6/bin/mongorestore
x mongodb-osx-x86_64-2.0.6/bin/mongos
x mongodb-osx-x86_64-2.0.6/bin/mongosniff
x mongodb-osx-x86_64-2.0.6/bin/mongostat
x mongodb-osx-x86_64-2.0.6/bin/mongotop
x mongodb-osx-x86_64-2.0.6/GNU-AGPL-3.0
x mongodb-osx-x86_64-2.0.6/README
x mongodb-osx-x86_64-2.0.6/THIRD-PARTY-NOTICES
```

To bootstrap MongoDB, you need to create a folder to contain the data and then start the *mongod* binary, pointing it to the just-created directory (see Example 6-3).

Example 6-3. Preparing and starting MongoDB

```
$ cd mongodb-osx-x86_64-2.0.6
$ mkdir data
$ ./bin/mongod --dbpath=data

Mon Jun 18 12:35:00 [initandlisten] MongoDB starting : pid=15216 port=27017 dbpath=data
  64-bit …
Mon Jun 18 12:35:00 [initandlisten] db version v2.0.6, pdfile version 4.5
Mon Jun 18 12:35:00 [initandlisten] git version: e1c0cbc25863f6356aa4e31375add7bb49fb05bc
Mon Jun 18 12:35:00 [initandlisten] build info: Darwin erh2.10gen.cc 9.8.0 Darwin Kernel
  Version 9.8.0: …
Mon Jun 18 12:35:00 [initandlisten] options: { dbpath: "data" }
Mon Jun 18 12:35:00 [initandlisten] journal dir=data/journal
Mon Jun 18 12:35:00 [initandlisten] recover : no journal files present, no recovery needed
```

```
Mon Jun 18 12:35:00 [websvr] admin web console waiting for connections on port 28017
Mon Jun 18 12:35:00 [initandlisten] waiting for connections on port 27017
```

As you can see, MongoDB starts up, uses the given path to store the data, and is now waiting for connections.

Using the MongoDB Shell

Let's explore the very basic operations of MongoDB using its shell. Switch to the directory in which you've just unzipped MongoDB and run the shell using the *mongo* binary, as shown in Example 6-4.

Example 6-4. Starting the MongoDB shell

```
$ cd ~/dev/mongodb-osx-x86_64-2.0.6
$ ./bin/mongo

MongoDB shell version: 2.0.6
connecting to: test
>
```

The shell will connect to the locally running MongoDB instance. You can now use the `show dbs` command to inspect all database, currently available on this instance. In Example 6-5, we select the local database and issue a `show collections` command, which will not reveal anything at this point because our database is still empty.

Example 6-5. Selecting a database and inspecting collections

```
> show dbs
local (empty)
> use local
switched to db local
> show collections
>
```

Now let's add some data to the database. We do so by using the `save(…)` command of a collection of our choice and piping the relevant data in JSON format to the function. In Example 6-6, we add two customers, Dave and Carter.

Example 6-6. Inserting data into MongoDB

```
> db.customers.save({ firstname : 'Dave', lastname : 'Matthews',
                      emailAddress : 'dave@dmband.com' })
> db.customers.save({ firstname : 'Carter', lastname : 'Beauford' })
> db.customers.find()
{ "_id" : ObjectId("4fdf07c29c62ca91dcdfd71c"), "firstname" : "Dave",
  "lastname" : "Matthews", "emailAddress" : "dave@dmband.com" }
{ "_id" : ObjectId("4fdf07da9c62ca91dcdfd71d"), "firstname" : "Carter",
  "lastname" : "Beauford" }
```

The `customers` part of the command identifies the collection into which the data will go. Collections will get created on the fly if they do not yet exist. Note that we've added

Carter without an email address, which shows that the documents can contain different sets of attributes. MongoDB will not enforce a schema onto you by default. The `find(…)` command actually can take a JSON document as input to create queries. To look up a customer with the email address of *dave@dmband.com*, the shell interaction would look something like Example 6-7.

Example 6-7. Looking up data in MongoDB

```
> db.customers.find({ emailAddress : 'dave@dmband.com' })
{ "_id" : ObjectId("4fdf07c29c62ca91dcdfd71c"), "firstname" : "Dave",
  "lastname" : "Matthews", "emailAddress" : "dave@dmband.com" }
```

You can find out more about working with the MongoDB shell at the MongoDB home page. Beyond that, [ChoDir10] is a great resource to dive deeper into the store's internals and how to work with it in general.

The MongoDB Java Driver

To access MongoDB from a Java program, you can use the Java driver provided and maintained by 10gen, the company behind MongoDB. The core abstractions to interact with a store instance are `Mongo`, `Database`, and `DBCollection`. The `Mongo` class abstracts the connection to a MongoDB instance. Its default constructor will reach out to a locally running instance on subsequent operations. As you can see in Example 6-8, the general API is pretty straightforward.

Example 6-8. Accessing a MongoDB instance through the Java driver

```
Mongo mongo = new Mongo();
DB database = mongo.getDb("database");
DBCollection customers = db.getCollection("customers");
```

This appears to be classical infrastructure code that you'll probably want to have managed by Spring to some degree, just like you use a `DataSource` abstraction when accessing a relational database. Beyond that, instantiating the `Mongo` object or working with the `DBCollection` subsequently could throw exceptions, but they are MongoDB-specific and shouldn't leak into client code. Spring Data MongoDB will provide this basic integration into Spring through some infrastructure abstractions and a Spring namespace to ease the setup even more. Read up on this in "Setting Up the Infrastructure Using the Spring Namespace" on page 81.

The core data abstraction of the driver is the `DBObject` interface alongside the `BasicDBObject` implementation class. It can basically be used like a plain Java `Map`, as you can see in Example 6-9.

Example 6-9. Creating a MongoDB document using the Java driver

```
DBObject address = new BasicDBObject("city", "New York");
address.put("street", "Broadway");
```

```
DBObject addresses = new BasicDBList();
addresses.add(address);

DBObject customer = new BasicDBObject("firstname", "Dave");
customer.put("lastname", "Matthews");
customer.put("addresses", addresses);
```

First, we set up what will end up as the embedded address document. We wrap it into a list, set up the basic customer document, and finally set the complex address property on it. As you can see, this is very low-level interaction with the store's data structure. If we wanted to persist Java domain objects, we'd have to map them in and out of `BasicDBObject`s manually—for each and every class. We will see how Spring Data MongoDB helps to improve that situation in a bit. The just-created document can now be handed to the `DBCollection` object to be stored, as shown in Example 6-10.

Example 6-10. Persisting the document using the MongoDB Java driver

```
DBCollection customers = db.getCollection("customers");
customers.insert(customer);
```

Setting Up the Infrastructure Using the Spring Namespace

The first thing Spring Data MongoDB helps us do is set up the necessary infrastructure to interact with a MongoDB instance as Spring beans. Using JavaConfig, we can simply extend the `AbstractMongoConfiguration` class, which contains a lot of the basic configuration for us but we can tweak to our needs by overriding methods. Our configuration class looks like Example 6-11.

Example 6-11. Setting up MongoDB infrastructure with JavaConfig

```
@Configuration
@EnableMongoRepositories
class ApplicationConfig extends AbstractMongoConfiguration {

  @Override
  protected String getDatabaseName() {
    return "e-store";
  }

  @Override
  public Mongo mongo() throws Exception {
    Mongo mongo = new Mongo();
    mongo.setWriteConcern(WriteConcern.SAFE);
    return
  }
}
```

We have to implement two methods to set up the basic MongoDB infrastructure. We provide a database name and a Mongo instance, which encapsulates the information about how to connect to the data store. We use the default constructor, which will

assume we have a MongoDB instance running on our local machine listening to the default port, 27017. Right after that, we set the `WriteConcern` to be used to `SAFE`. The `WriteConcern` defines how long the driver waits for the MongoDB server when doing write operations. The default setting does not wait at all and doesn't complain about network issues or data we're attempting to write being illegal. Setting the value to `SAFE` will cause exceptions to be thrown for network issues and makes the driver wait for the server to okay the written data. It will also generate complaints about index constraints being violated, which will come in handy later.

These two configuration options will be combined in a bean definition of a `SimpleMongoDbFactory` (see the `mongoDbFactory()` method of `AbstractMongoConfiguration`). The `MongoDbFactory` is in turn used by a `MongoTemplate` instance, which is also configured by the base class. It is the central API to interact with the MongoDB instance, and persist and retrieve objects from the store. Note that the configuration class you find in the sample project already contains extended configuration, which will be explained later.

The XML version of the previous configuration looks as follows like Example 6-12.

Example 6-12. Setting up MongoDB infrastructure using XML

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:mongo="http://www.springframework.org/schema/data/mongo"
       xsi:schemaLocation="http://www.springframework.org/schema/data/mongo
                           http://www.springframework.org/schema/data/mongo/
                               spring-mongo.xsd
                           http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

  <mongo:db-factory id="mongoDbFactory" dbname="e-store" />

  <bean id="mongoTemplate" class="org.springframework.data.mongodb.core.MongoTemplate">
    <constructor-arg ref="mongoDbFactory" />
    <property name="writeConcern" value="SAFE" />
  </bean>

</beans>
```

The `<db-factory />` element sets up the `SimpleMongoDbFactory` in a similar way as we saw in the JavaConfig example. The only difference here is that it also defaults the Mongo instance to be used to the one we had to configure manually in JavaConfig. We can customize that setup by manually defining a `<mongo:mongo />` element and configuring its attributes to the values needed. As we'd like to avoid that here, we set the `WriteConcern` to be used on the `MongoTemplate` directly. This will cause all write operations invoked through the template to be executed with the configured concern.

The Mapping Subsystem

To ease persisting objects, Spring Data MongoDB provides a mapping subsystem that can inspect domain classes for persistence metadata and automatically convert these objects into MongoDB `DBObject`s. Let's have a look at the way our domain model could be modeled and what metadata is necessary to tweak the object-document mapping to our needs.

The Domain Model

First, we introduce a base class for all of our top-level documents, as shown in Example 6-13. It consists of an `id` property only and thus removes the need to repeat that property all over the classes that will end up as documents. The `@Id` annotation is optional. By default we consider properties named `id` or `_id` the ID field of the document. Thus, the annotation comes in handy in case you'd like to use a different name for the property or simply to express a special purpose for it.

Example 6-13. The AbstractDocument class

```
public class AbstractDocument {

  @Id
  private BigInteger id;

  …
}
```

Our `id` property is of type `BigInteger`. While we generally support any type to be used as `id`, there are a few types that allow special features to be applied to the document. Generally, the recommended `id` type to end up in the persistent document is `ObjectID`. `ObjectID`s are value objects that allow for generating consistently growing `id`s even in a cluster environment. Beyond that, they can be autogenerated by MongoDB. Translating these recommendations into the Java driver world also implies it's best to have an `id` of type `ObjectID`. Unfortunately, this would create a dependency to the Mongo Java driver inside your domain objects, which might be something you'd like to avoid. Because `ObjectID`s are 12-byte binary values essentially, they can easily be converted into either `String` or `BigInteger` values. So, if you're using `String`, `BigInteger`, or `ObjectID` as `id` types, you can leverage MongoDB's `id` autogeneration feature, which will automatically convert the `id` values into `ObjectID`s before persisting and back when reading. If you manually assign `String` values to your `id` fields that cannot be converted into an `ObjectID`, we will store them as is. All other types used as `id` will also be stored this way.

Addresses and email addresses

The `Address` domain class, shown in Example 6-14, couldn't be simpler. It's a plain wrapper around three `final` primitive `String` values. The mapping subsystem will

transform objects of this type into a `DBObject` by using the property names as field keys and setting the values appropriately, as you can see in Example 6-15.

Example 6-14. The Address domain class

```
public class Address {

  private final String street, city, country;

  public Address(String street, String city, String country) {

    Assert.hasText(street, "Street must not be null or empty!");
    Assert.hasText(city, "City must not be null or empty!");
    Assert.hasText(country, "Country must not be null or empty!");

    this.street = street;
    this.city = city;
    this.country = country;
  }

  // … additional getters
}
```

Example 6-15. An Address object's JSON representation

```
{ street : "Broadway",
  city : "New York",
  country : "United States" }
```

As you might have noticed, the `Address` class uses a complex constructor to prevent an object from being able to be set up in an invalid state. In combination with the `final` fields, this makes up a classic example of a value object that is immutable. An `Address` will never be changed, as changing a property forces a new `Address` instance to be created. The class does not provide a no-argument constructor, which raises the question of how the object is being instantiated when the `DBObject` is read from the database and has to be turned into an `Address` instance. Spring Data uses the concept of a so-called *persistence constructor*, the constructor being used to instantiate persisted objects. Your class providing a no-argument constructor (either implicit or explicit) is the easy scenario. The mapping subsystem will simply use that to instantiate the entity via reflection. If you have a constructor taking arguments, it will try to match the parameter names against property names and eagerly pull values from the store representation—the `DBObject` in the case of MongoDB.

Another example of a domain concept embodied through a value object is an `EmailAddress` (Example 6-16). Value objects are an extremely powerful way to encapsulate business rules in code and make the code more expressive, readable, testable, and maintainable. For a more in-depth discussion, refer to Dan Bergh-Johnsson's talk on this topic, available on InfoQ. If you carried an email address around in a plain `String`, you could never be sure whether it had been validated and actually represents a valid email address. Thus, the plain wrapper class checks the given source value

against a regular expression and rejects it right away if it doesn't match the expression. This way, clients can be sure they're dealing with a proper email address if they get hold of an `EmailAddress` instance.

Example 6-16. The EmailAddress domain class

```
public class EmailAddress {

  private static final String EMAIL_REGEX = …;
  private static final Pattern PATTERN = Pattern.compile(EMAIL_REGEX);

  @Field("email")
  private final String value;

  public EmailAddress(String emailAddress) {
    Assert.isTrue(isValid(emailAddress), "Invalid email address!");
    this.value = emailAddress;
  }

  public static boolean isValid(String source) {
    return PATTERN.matcher(source).matches();
  }
}
```

The `value` property is annotated with `@Field`, which allows for customizing the way a property is mapped to a field in a `DBObject`. In our case, we map the rather generic `value` to a more specific `email`. While we could have simply named the property `email` in the first place in our situation, this feature comes in handy in two major scenarios. First, say you want to map classes onto existing documents that might have chosen field keys that you don't want to let leak into your domain objects. `@Field` generally allows decoupling between field keys and property names. Second, in contrast to the relational model, field keys are repeated for every document, so they can make up a large part of the document data, especially if the values you store are small. So you could reduce the space required for keys by defining rather short ones to be used, with the trade-off of slightly reduced readability of the actual JSON representation.

Now that we've set the stage with our basic domain concept implementations, let's have a look at the classes that actually will make up our documents.

Customers

The first thing you'll probably notice about the `Customer` domain class, shown in Example 6-17, is the `@Document` annotation. It is actually an optional annotation to some degree. The mapping subsystem would still be able to convert the class into a `DBObject` if the annotation were missing. So why do we use it here? First, we can configure the mapping infrastructure to scan for domain classes to be persisted. This will pick up only classes annotated with `@Document`. Whenever an object of a type currently unknown to the mapping subsystem is handed to it, the subsystem automatically and immediately inspects the class for mapping information, slightly decreasing the per-

formance of that very first conversion operation. The second reason to use `@Document` is the ability to customize the MongoDB collection in which a domain object is stored. If the annotation is not present at all or the collection attribute is not configured, the collection name will be the simple class name with the first letter lowercased. So, for example, a `Customer` would go into the `customer` collection.

The code might look slightly different in the sample project, because we're going to tweak the model slightly later to improve the mapping. We'd like to keep it simple at this point to ease your introduction, so we will concentrate on general mapping aspects here.

Example 6-17. The Customer domain class

```
@Document
public class Customer extends AbstractDocument {

  private String firstname, lastname;

  @Field("email")
  private EmailAddress emailAddress;
  private Set<Address> addresses = new HashSet<Address>();

  public Customer(String firstname, String lastname) {

    Assert.hasText(firstname);
    Assert.hasText(lastname);

    this.firstname = firstname;
    this.lastname = lastname;
  }

  // additional methods and accessors
}
```

The `Customer` class contains two primitive properties to capture first name and last name as well as a property of our `EmailAddress` domain class and a `Set` of `Addresses`. The `emailAddress` property is annotated with `@Field`, which (as noted previously) allows us to customize the key to be used in the MongoDB document.

Note that we don't actually need any annotations to configure the relationship between the `Customer` and `EmailAddress` and the `Addresses`. This is mostly driven from the fact that MongoDB documents can contain complex values (i.e., nested documents). This has quite severe implications for the class design and the persistence of the objects. From a design point of view, the `Customer` becomes an *aggregate root*, in the domain-driven design terminology. `Addresses` and `EmailAddresses` are never accessed individually but rather through a `Customer` instance. We essentially model a tree structure here that maps nicely onto MongoDB's document model. This results in the object-to-document mapping being much less complicated than in an object-relational scenario. From a persistence point of view, storing the entire `Customer` alongside its `Addresses`

and `EmailAddress`es becomes a single—and thus atomic—operation. In a relational world, persisting this object would require an insert for each `Address` plus one for the `Customer` itself (assuming we'd inline the `EmailAddress` into a column of the table the `Customer` ends up in). As the rows in the table are only loosely coupled to each other, we have to ensure the consistency of the insert by using a transaction mechanism. Beyond that, the insert operations have to be ordered correctly to satisfy the foreign key relationships.

However, the document model not only has implications on the writing side of persistence operations, but also on the reading side, which usually makes up even more of the access operations for data. Because the document is a self-contained entity in a collection, accessing it does not require reaching out into other collections, documents or the like. Speaking in relational terms, a document is actually a set of prejoined data. Especially if applications access data of a particular granularity (which is what is usually driving the class design to some degree), it hardly makes sense to tear apart the data on writes and rejoin it on each and every read. A complete customer document would look something like Example 6-18.

Example 6-18. Customer document

```
{ firstname : "Dave",
  lastname : "Matthews",
  email : { email : "dave@dmband.com" },
  addresses : [ { street : "Broadway",
                  city : "New York",
                  country : "United States" } ] }
```

Note that modeling an email address as a value object requires it to be serialized as a nested object, which essentially duplicates the key and makes the document more complex than necessary. We'll leave it as is for now, but we'll see how to improve it in "Customizing Conversion" on page 91.

Products

The `Product` domain class (Example 6-19) again doesn't contain any huge surprises. The most interesting part probably is that `Map`s can be stored natively—once again due to the nature of the documents. The attributes will be just added as a nested document with the `Map` entries being translated into document fields. Note that currently, only `String`s can be used as `Map` keys.

Example 6-19. The Product domain class

```
@Document
public class Product extends AbstractDocument {

  private String name, description;
  private BigDecimal price;
  private Map<String, String> attributes = new HashMap<String, String>();
```

```
  // … additional methods and accessors
}
```

Orders and line items

Moving to the order subsystem of our application, we should look first at the `LineItem` class, shown in Example 6-20.

Example 6-20. The LineItem domain class

```
public class LineItem extends AbstractDocument {

  @DBRef
  private Product product;
  private BigDecimal price;
  private int amount;

  // … additional methods and accessors
}
```

First we see two basic properties, `price` and `amount`, declared without further mapping annotations because they translate into document fields natively. The `product` property, in contrast, is annotated with `@DBRef`. This will cause the `Product` object inside the `LineItem` to not be embedded. Instead, there will be a pointer to a document in the collection that stores `Product`s. This is very close to a foreign key in the world of relational databases.

Note that when we're storing a `LineItem`, the `Product` instance referenced has to be saved already—so currently, there's no cascading of save operations available. When we're reading `LineItem`s from the store, the reference to the `Product` will be resolved eagerly, causing the referenced document to be read and converted into a `Product` instance.

To round things off, the final bit we should have a look at is the `Order` domain class (Example 6-21).

Example 6-21. The Order domain class

```
@Document
public class Order extends AbstractDocument {

  @DBRef
  private Customer customer;
  private Address billingAddress;
  private Address shippingAddress;
  private Set<LineItem> lineItems = new HashSet<LineItem>();

  // - additional methods and parameters
}
```

Here we essentially find a combination of mappings we have seen so far. The class is annotated with `@Document` so it can be discovered and inspected for mapping informa-

tion during application context startup. The `Customer` is referenced using an `@DBRef`, as we'd rather point to one than embedding it into the document. The `Address` properties and the `LineItem`s are embedded as is.

Setting Up the Mapping Infrastructure

As we've seen how the domain class is persisted, now let's have a look at how we actually set up the mapping infrastructure to work for us. In most cases this is pretty simple, and some of the components that use the infrastructure (and which we'll introduce later) will fall back to reasonable default setups that generally enable the mapping subsystem to work as just described. However, if you'd like to customize the setup, you'll need to tweak the configuration slightly. The two core abstractions that come into play here are the `MongoMappingContext` and `MappingMongoConverter`. The former is actually responsible for building up the domain class metamodel to avoid reflection lookups (e.g., to detect the `id` property or determine the field key on each and every persistence operation). The latter is actually performing the conversion using the mapping information provided by the `MappingContext`. You can simply use these two abstractions together to trigger object-to-DBObject-and-back conversion programmatically (see Example 6-22).

Example 6-22. Using the mapping subsystem programmatically

```
MongoMappingContext context = new MongoMappingContext();
MongoDbFactory dbFactory = new SimpleMongoDbFactory(new Mongo(), "database");
MappingMongoConverter converter = new MongoMappingContext(dbFactory, context);

Customer customer = new Customer("Dave", "Matthews");
customer.setEmailAddress(new EmailAddress("dave@dmband.com"));
customer.add(new Address("Broadway", "New York", "United States"));

DBObject sink = new BasicDBObject();
converter.write(customer, sink);

System.out.println(sink.toString());
```

```
{ firstname : "Dave",
  lastname : "Matthews",
  email : { email : "dave@dmband.com" },
  addresses : [ { street : "Broadway",
                  city : "New York",
                  country : "United States" } ] }
```

We set up instances of a `MongoMappingContext` as well as a `SimpleMongoDbFactory`. The latter is necessary to potentially load `@DBRef` annotated documents eagerly. This is not needed in our case, but we still have to set up the `MongoMappingConverter` instance correctly. We then set up a `Customer` instance as well as a `BasicDBObject` and invoke the converter to do its work. After that, the `DBObject` is populated with the data as expected.

Using the Spring namespace

The Spring namespace that ships with Spring Data MongoDB contains a `<mongo:mapping-converter />` element that basically sets up an instance of `MappingMongoConverter` as we've seen before. It will create a `MongoMappingContext` internally and expect a Spring bean named `mongoDbFactory` in the `ApplicationContext`. We can tweak this by using the `db-factory-ref` attribute of the namespace element. See Example 6-23.

Example 6-23. Setting up a MappingMongoConverter in XML

```
<mongo:mapping-converter id="mongoConverter"
                         base-package="com.oreilly.springdata.mongodb" />
```

This configuration snippet configures the `MappingMongoConverter` to be available under the `id` `mongoConverter` in the Spring application context. We point the `base-package` attribute to our project's base package to pick up domain classes and build the persistence metadata at application context startup.

In Spring JavaConfig

To ease the configuration when we're working with Spring JavaConfig classes, Spring Data MongoDB ships with a configuration class that declares the necessary infrastructure components in a default setup and provides callback methods to allow us to tweak them as necessary. To mimic the setup just shown, our configuration class would have to look like Example 6-24.

Example 6-24. Basic MongoDB setup with JavaConfig

```
@Configuration
class ApplicationConfig extends AbstractMongoConfiguration {

  @Override
  protected String getDatabaseName() {
    return "e-store";
  }

  @Override
  public Mongo mongo() throws Exception {

    Mongo mongo = new Mongo();
    mongo.setWriteConcern(WriteConcern.SAFE);
    return mongo;
  }

  @Override
  protected String getMappingBasePackage() {
    return "com.oreilly.springdata.mongodb"
  }
}
```

The first two methods are required to be implemented by the superclass because it sets up a `SimpleMongoDbFactory` to access a MongoDB already. Beyond these necessary

implementations, we override the `getMappingBasePackage()` method to indicate that the mapping subsystem will inspect this package, and all below it, for the classes annotated with `@Document`. This is not strictly necessary, as the mapping infrastructure will scan the package of the configuration class by default. We just list it here to demonstrate how it could be reconfigured.

Indexing

MongoDB supports indexes just as a relational store does. You can configure the index programatically or by using a mapping annotation. Because we're usually going to retrieve `Customer`s by their email addresses, we'd like to index on those. Thus, we add the `@Index` annotation to the `emailAddress` property of the `Customer` class, as shown in Example 6-25.

Example 6-25. Configuring an index on the emailAddress property of Customer

```
@Document
public class Customer extends AbstractDocument {

  @Index(unique = true)
  private EmailAddress emailAddress;

  …
}
```

We'd like to prevent duplicate email addresses in the system, so we set the `unique` flag to `true`. This will cause MongoDB to prevent `Customer`s from being created or updated with the same email address as another `Customer`. We can define indexes including multiple properties by using the `@CompoundIndex` annotation on the domain class.

Index metadata will be discovered when the class is discovered by the `MappingContext`. As the information is stored alongside the collection, to which the class gets persisted, it will get lost if you drop the collection. To avoid that, remove all documents from the collection.

You can find an example use case of a domain object being rejected in the `CustomerRepositoryIntegrationTests` class of the sample application. Note that we expect a `DuplicateKeyException` to be thrown, as we persist a second customer with the email address obtained from an already existing one.

Customizing Conversion

The mapping subsystem provides a generic way to convert your Java objects into MongoDB `DBObject`s and vice versa. However, you might want to manually implement a conversion of a given type. For example, you've seen previously that the introduction of a value object to capture email addresses resulted in a nested document that you

might want to avoid to keep the document structure simple, especially since we can simply inline the `EmailAddress` value into the customer object. To recap the scenario, Example 6-26 shows where we'd like to start.

Example 6-26. The Customer class and its DBObject representation

```
@Document
public class Customer extends AbstractDocument {

  private String firstname, lastname;

  @Field("email")
  private EmailAddress emailAddress;

  …
}
```

```
{ firstname : "Dave",
  lastname : "Matthews",
  email : { email : "dave@dmband.com" }, … }
```

What we would actually like to end up with is a simpler document looking something like Example 6-27.

Example 6-27. The intended document structure of a Customer

```
{ firstname : "Dave",
  lastname : "Matthews",
  email : "dave@dmband.com", … }
```

Implementing custom converters

The mapping subsystem allows you to manually implement the object-to-document-and-back conversion yourself by leveraging the Spring conversion service's `Converter` abstraction. Since we'd like to turn the complex object into a plain `String`, we essentially need to implement a writing `Converter<EmailAddress, String>` as well as one to construct `EmailAddress` objects from `String`s (i.e., a `Converter<String, EmailAddress>`), as shown in Example 6-28.

Example 6-28. Custom Converter implementations for EmailAddress

```
@Component
class EmailAddressToStringConverter implements Converter<EmailAddress, String> {

  public String convert(EmailAddress source) {
    return source == null ? null : source.value;
  }
}

@Component
class StringToEmailAddressConverter implements Converter<String, EmailAddress> {

  public EmailAddress convert(String source) {
    return StringUtils.hasText(source) ? new EmailAddress(source) : null;
```

```
  }
}
```

Registering custom converters

The just-implemented converters now have to be registered with the mapping subsystem. Both the Spring XML namespace as well as the provided Spring JavaConfig configuration base class make this very easy. In the XML world, it's just a matter of declaring a nested element inside `<mongo:mapping-converter />` and activating component scanning by setting the `base-package` attribute. See Example 6-29.

Example 6-29. Registering custom converters with the XML namespace

```
<mongo:mapping-converter id="mongoConverter" base-package="com.oreilly.springdata.mongodb">
  <mongo:custom-converters base-package="com.oreilly.springdata.mongodb" />
</mongo:mapping-converter>
```

In the JavaConfig world, the configuration base class provides a callback method for you to return an instance of `CustomConversions`. This class is a wrapper around the `Converter` instances you hand it, which we can inspect later to configure the `MappingContext` and `MongoConverter` appropriately, as well as the `ConversionService` to eventually perform the conversions. In Example 6-30, we access the `Converter` instances by enabling component scanning and autowiring them into the configuration class to eventually wrap them into the `CustomConversions` instance.

Example 6-30. Registering custom converters using Spring JavaConfig

```
@Configuration
@ComponentScan
class ApplicationConfig extends AbstractMongoConfiguration {

  @Autowired
  private List<Converter<?, ?>> converters;

  @Override
  public CustomConversions customConversions() {
    return new CustomConversions(converters);
  }
}
```

If we now obtain a `MappingMongoConverter` from the application context and invoke a conversion, as demonstrated in Example 6-22, the output would change to that shown in Example 6-31.

Example 6-31. Document structure with custom converters for EmailAddress applied

```
{ firstname : "Dave",
  lastname : "Matthews",
  email : "dave@dmband.com", … }
```

MongoTemplate

Now that we have both the general infrastructure in place and understand the way that object mapping works and can be configured, let's continue with the API we provide to interact with the store. As with all the other Spring Data modules, the core of the API is the `MongoOperations` interface, backed by a `MongoTemplate` implementation. Template implementations in Spring serve two primary purposes: resource management and exception translation. This means that `MongoTemplate` will take care of acquiring a connection through the configured `MongoDbFactory` and clean it up properly after the interaction with the store has ended or an exception has occurred. Exceptions being thrown by MongoDB will be transparently converted into Spring's `DataAccessException` hierarchy to prevent the clients from having to know about the persistence technology being used.

To illustrate the usage of the API, we will look at a repository implementation of the `CustomerRepository` interface (Example 6-32). It's called `MongoDbCustomerRepository` and located in the `com.oreilly.springdata.mongodb.core` package.

Example 6-32. MongoDbCustomerRepository implementation

```
import static org.springframework.data.mongodb.core.query.Criteria.*;
import static org.springframework.data.mongodb.core.query.Query.*;
…

@Repository
@Profile("mongodb")
class MongoDbCustomerRepository implements CustomerRepository {

  private final MongoOperations operations;

  @Autowired
  public MongoDbCustomerRepository(MongoOperations operations) {
    Assert.notNull(operations);
    this.operations = operations;
  }

  @Override
  public Customer findOne(Long id) {

    Query query = query(where("id").is(id));
    return operations.findOne(query, Customer.class);
  }

  @Override
  public Customer save(Customer customer) {

    operations.save(customer);
    return customer;
  }

  @Override
  public Customer findByEmailAddress(EmailAddress emailAddress) {
```

```
    Query query = query(where("emailAddress").is(emailAddress));
    return operations.findOne(query, Customer.class);
  }
}
```

As you can see, we have a standard Spring component annotated with `@Repository` to make the class discoverable by classpath scanning. We add the `@Profile` annotation to make sure it will be activated only if the configured Spring profile is activated. This will prevent the class from leaking into the default bean setup, which we will use later when introducing the Spring Data repositories for MongoDB.

The class's only dependency is `MongoOperations`; this is the interface of `MongoTemplate`, which we have configured in our application context (see *application-context.xml* or `ApplicationConfig` class [Example 6-12]). `MongoTemplate` provides two categories of methods to be used:

- General-purpose, high-level methods that enable you to execute commonly needed operations as one-line statements. This includes basic functions like `findOne(…)`, `findAll(…)`, `save(…)`, and `delete(…)`; more MongoDB-specific ones like `updateFirst(…)`, `updateMulti(…)`, and `upsert(…)`; and map-reduce and geospatial operations like `mapReduce(…)` and `geoNear(…)`. All these methods automatically apply the object-to-store mapping discussed in "The Mapping Subsystem" on page 83.
- Low-level, callback-driven methods that allow you to interact with the MongoDB driver API in the even that the high-level operations don't suffice for the functionality you need. These methods all start with `execute` and take either a `CollectionCallback` (providing access to a MongoDb `DbCollection`), `DbCallback` (providing access to a MongoDB `DB`), or a `DocumentCallbackHandler` (to process a `DBObject` directly).

The simplest example of the usage of the high-level API is the `save(…)` method of `MongoDbCustomerRepository`. We simply use the `save(…)` method of the `MongoOperations` interface to hand it the provided `Customer`. It will in turn convert the domain object into a MongoDB `DBObject` and save that using the MongoDB driver API.

The two other methods in the implementation—`findOne(…)` and `findByEmailAddress(…)`—use the query API provided by Spring Data MongoDB to ease creating queries to access MongoDB documents. The `query(…)` method is actually a static factory method of the `Query` class statically imported at the very top of the class declaration. The same applies to the `where(…)` method, except it's originating from `Criteria`. As you can see, defining a query is remarkably simple. Still, there are a few things to notice here.

In `findByEmailAddress(…)`, we reference the `emailAddress` property of the `Customer` class. Because it has been mapped to the `email` key by the `@Field` annotation, the property reference will be automatically translated into the correct field reference. Also, we hand the plain `EmailAddress` object to the criteria to build the equality predicate. It will also be transformed by the mapping subsystem before the query is actually applied. This

includes custom conversions registered for the given type as well. Thus, the `DBObject` representing the query will look something like Example 6-33.

Example 6-33. The translated query object for findByEmailAddress(…)

```
{ "email" : "dave@dmband.com" }
```

As you can see, the field key was correctly translated to `email` and the value object properly inlined due to the custom converter for the `EmailAddress` class we introduced in "Implementing custom converters" on page 92.

Mongo Repositories

As just described, the `MongoOperations` interface provides a decent API to implement a repository manually. However, we can simplify this process even further using the Spring Data repository abstraction, introduced in Chapter 2. We'll walk through the repository interface declarations of the sample project and see how invocations to the repository methods get handled.

Infrastructure Setup

We activate the repository mechanism by using either a JavaConfig annotation (Example 6-34) or an XML namespace element.

Example 6-34. Activating Spring Data MongoDB repositories in JavaConfig

```
@Configuration
@ComponentScan(basePackageClasses = ApplicationConfig.class)
@EnableMongoRepositories
public class ApplicationConfig extends AbstractMongoConfiguration {

  …
}
```

In this configuration sample, the `@EnableMongoRepositories` is the crucial part. It will set up the repository infrastructure to scan for repository interfaces in the package of the annotated configuration class by default. We can alter this by configuring either the `basePackage` or `basePackageClasses` attributes of the annotation. The XML equivalent looks very similar, except we have to configure the base package manually (Example 6-35).

Example 6-35. Activating Spring Data MongoDB repositories in XML

```
<mongo:repositories base-package="com.oreilly.springdata.mongodb" />
```

Repositories in Detail

For each of the repositories in the sample application, there is a corresponding integration test that we can run against a local MongoDB instance. These tests interact with the repository and invoke the methods exposed. With the log level set to `DEBUG`, you should be able to follow the actual discovery steps, query execution, etc.

Let's start with `CustomerRepository` since it's the most basic one. It essentially looks like Example 6-36.

Example 6-36. CustomerRepository interface

```
public interface CustomerRepository extends Repository<Customer, Long> {

  Customer findOne(Long id);

  Customer save(Customer customer);

  Customer findByEmailAddress(EmailAddress emailAddress);
}
```

The first two methods are essentially CRUD methods and will be routed into the generic `SimpleMongoRepository`, which provides the declared methods. The general mechanism for that is discussed in Chapter 2. So the really interesting method is `findByEmailAddress(…)`. Because we don't have a manual query defined, the query derivation mechanism will kick in, parse the method, and derive a query from it. Since we reference the `emailAddress` property, the logical query derived is essentially `emailAddress = ?0`. Thus, the infrastructure will create a `Query` instance using the Spring Data MongoDB query API. This will in turn translate the property reference into the appropriate field mapping so that we end up using the query `{ email : ?0 }`. On method invocation, the given parameters will be bound to the query and executed eventually.

The next repository is `PersonRepository`, shown in Example 6-37.

Example 6-37. PersonRepository interface

```
public interface ProductRepository extends CrudRepository<Product, Long> {

  Page<Product> findByDescriptionContaining(String description, Pageable pageable);

  @Query("{ ?0 : ?1 }")
  List<Product> findByAttributes(String key, String value);
}
```

The first thing to notice is that `ProductRepository` extends `CrudRepository` instead of the plain `Repository` marker interface. This causes the CRUD methods to be pulled into our repository definition. Thus, we don't have to manually declare `findOne(…)` and `save(…)` manually. `findByDescriptionContaining(…)` once again uses the query derivation mechanism, just as we have seen in `CustomerRepository.findByEmailAddress(…)`. The difference from the former method is that this one additionally qualifies the

predicate with the `Containing` keyword. This will cause the description parameter handed into the method call to be massaged into a regular expression to match descriptions that contain the given `String` as a substring.

The second thing worth noting here is the use of the pagination API (introduced in "Pagination and Sorting" on page 18). Clients can hand in a `Pageable` to the method to restrict the results returned to a certain page with a given number and page size, as shown in Example 6-38. The returned `Page` then contains the results plus some meta-information, such as about how many pages there are in total. You can see a sample usage of the method in `PersonRepositoryIntegrationTests`: the `lookupProductsByDescription()` method.

Example 6-38. Using the findByDescriptionContaining(…) method

```
Pageable pageable = new PageRequest(0, 1, Direction.DESC, "name");
Page<Product> page = repository.findByDescriptionContaining("Apple", pageable);

assertThat(page.getContent(), hasSize(1));
assertThat(page, Matchers.<Product> hasItems(named("iPad")));
assertThat(page.isFirstPage(), is(true));
assertThat(page.isLastPage(), is(false));
assertThat(page.hasNextPage(), is(true));
```

First, we set up a `PageRequest` to request the first page with a page size of 1, requiring the results to be sorted in descending order by name. See how the returned page provides not only the results, but also information on where the returned page is located in the global set of pages.

The second method (refer back to Example 6-37) declared uses the `@Query` annotation to manually define a MongoDB query. This comes in handy if the query derivation mechanism does not provide the functionality you need for the query, or the query method's name is awkwardly long. We set up a general query `{ ?0 : ?1 }` to bind the first argument of the method to act as key and the second one to act as value. The client can now use this method the query for `Product`s that have a particular attribute (e.g., a dock connector plug), as shown in Example 6-39.

Example 6-39. Querying for Products with a dock connector plug

```
List<Product> products = repository.findByAttributes("attributes.connector", "plug");

assertThat(products, Matchers.<Product> hasItems(named("Dock")));
```

As expected, the iPod dock is returned from the method call. This way, the business logic could easily implement `Product` recommendations based on matching attribute pairs (connector plug and socket).

Last but not least, let's have a look at the `OrderRepository` (Example 6-40). Given that we already discussed two repository interfaces, the last one shouldn't come with too many surprises.

Example 6-40. OrderRepository interface

```
public interface OrderRepository extends PagingAndSortingRepository<Order, Long> {

  List<Order> findByCustomer(Customer customer);
}
```

The query method declared here is just a straightforward one using the query derivation mechanism. What has changed compared to the previous repository interfaces is the base interface we extend from. Inheriting from `PagingAndSortingRepository` not only exposes CRUD methods, but also methods like `findAll(Pageable pageable)` allow for paginating the entire set of `Order`s in a convenient way. For more information on the pagination API in general, see "Pagination and Sorting" on page 18.

Mongo Querydsl Integration

Now that we've seen how to add query methods to repository interfaces, let's have a look at how we can use Querydsl to dynamically create predicates for entities and execute them via the repository abstraction. Chapter 3 provides a general introduction to what Querydsl actually is and how it works. If you've read "Repository Querydsl Integration" on page 51, you'll see how remarkably similar the setup and usage of the API is, although we query a totally different store.

To generate the metamodel classes, we have configured the Querydsl Maven plug-in in our *pom.xml* file, as shown in Example 6-41.

Example 6-41. Setting up the Querydsl APT processor for MongoDB

```
<plugin>
  <groupId>com.mysema.maven</groupId>
  <artifactId>maven-apt-plugin</artifactId>
  <version>1.0.5</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>process</goal>
      </goals>
      <configuration>
        <outputDirectory>target/generated-sources</outputDirectory>
        <processor>…data.mongodb.repository.support.MongoAnnotationProcessor</processor>
      </configuration>
    </execution>
  </executions>
</plugin>
```

The only difference from the JPA approach is that we configure a `MongoAnnotationProcessor`. It will configure the APT processor to inspect the annotations provided by the Spring Data MongoDB mapping subsystem to generate the metamodel correctly. Beyond that, we provide integration to let Querydsl consider our mappings—and thus potentially registered custom converters—when creating the MongoDB queries.

To include the API to execute predicates built with the generated metamodel classes, we let the `ProductRepository` additionally extend `QueryDslPredicateExecutor` (Example 6-42).

Example 6-42. The ProductRepository interface extending QueryDslPredicateExecutor

```
public interface ProductRepository extends CrudRepository<Product, Long>,
                                           QueryDslPredicateExecutor<Product> { … }
```

The `QuerydslProductRepositoryIntegrationTest` now shows how to make use of the predicates. Again, the code is pretty much a 1:1 copy of the JPA code. We obtain a reference iPad by executing the `product.name.eq("iPad")` predicate and use that to verify the result of the execution of the predicate, looking up products by description, as shown in Example 6-43.

Example 6-43. Using Querydsl predicates to query for Products

```
QProduct product = QProduct.product;

Product iPad = repository.findOne(product.name.eq("iPad"));
Predicate tablets = product.description.contains("tablet");

Iterable<Product> result = repository.findAll(tablets);
assertThat(result, is(Matchers.<Product> iterableWithSize(1)));
assertThat(result, hasItem(iPad));
```

CHAPTER 7

Neo4j: A Graph Database

Graph Databases

This chapter introduces an interesting kind of NoSQL store: graph databases. Graph databases are clearly post-relational data stores, because they evolve several database concepts much further while keeping other attributes. They provide the means of storing semistructured but highly connected data efficiently and allow us to query and traverse the linked data at a very high speed.

Graph data consists of nodes connected with directed and labeled relationships. In property graphs, both nodes and relationships can hold arbitrary key/value pairs. Graphs form an intricate network of those elements and encourage us to model domain and real-world data close to the original structure. Unlike relational databases, which rely on fixed schemas to model data, graph databases are schema-free and put no constraints onto the data structure. Relationships can be added and changed easily, because they are not part of a schema but rather part of the actual data.

We can attribute the high performance of graph databases to the fact that moving the cost of relating entities (joins) to the insertion time—by materializing the relationships as first-level citizens of the data structure—allows for constant time traversal from one entity (node) to another. So, regardless of the dataset size, the time for a given traversal across the graph is always determined by the number of hops in that traversal, not the number of nodes and relationships in the graph as a whole. In other database models, the cost of finding connections between two (or more) entities occurs on each query instead.

Thanks to this, a single graph can store many different domains, creating interesting connections between entities from all of them. Secondary access or index structures can be integrated into the graph to allow special grouping or access paths to a number of nodes or subgraphs.

Due to the nature of graph databases, they don't rely on aggregate bounds to manage atomic operations but instead build on the well-established transactional guarantees of an ACID (atomicity, consistency, isolation, durability) data store.

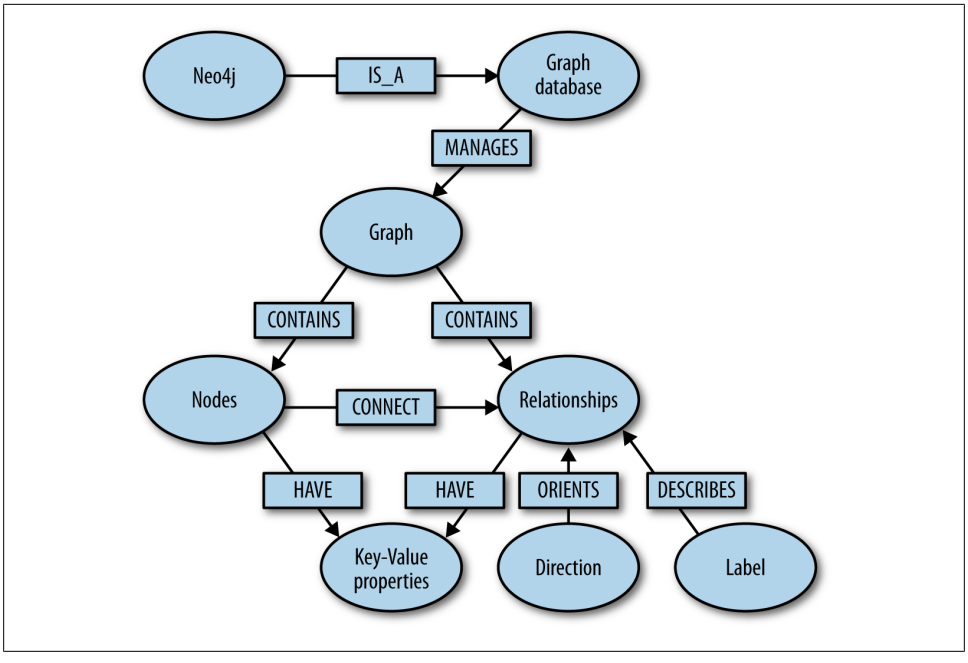


Figure 7-1. Graph database overview

Neo4j

Neo4j is the leading implementation of a property graph database. It is written predominantly in Java and leverages a custom storage format and the facilities of the Java Transaction Architecture (JTA) to provide XA transactions. The Java API offers an object-oriented way of working with the nodes and relationships of the graph (show in the example). Traversals are expressed with a fluent API. Being a graph database, Neo4j offers a number of graph algorithms like shortest path, Dijkstra, or A* out of the box.

Neo4j integrates a transactional, pluggable indexing subsystem that uses Lucene as the default. The index is used primarily to locate starting points for traversals. Its second use is to support unique entity creation. To start using Neo4j's embedded Java database, add the `org.neo4j:neo4j:<version>` dependency to your build setup, and you're ready to go. Example 7-1 lists the code for creating nodes and relationships with properties within transactional bounds. It shows how to access and read them later.

Example 7-1. Neo4j Core API Demonstration

```
GraphDatabaseService gdb = new EmbeddedGraphDatabase("path/to/database");

Transaction tx=gdb.beginTx();
try {
    Node dave = gdb.createNode();
    dave.setProperty("email","dave@dmband.com");
    gdb.index().forNodes("Customer").add
```

```
   (dave,"email",dave.getProperty("email");

    Node iPad = gdb.createNode();
    iPad.setProperty("name","Apple iPad");

    Relationship rel=dave.createRelationshipTo(iPad,Types.RATED);
    rel.setProperty("stars",5);

    tx.success();
} finally {
    tx.finish();
}

// to access the data

Node dave = gdb.index().forNodes("Customer").get("email","david@dmband.com").getSingle();
for (Relationship rating : dave.getRelationships(Direction.OUTGOING, Types.RATED)) {
    aggregate(rating.getEndNode(), rating.getProperty("stars"));
}
```

With the declarative Cypher query language, Neo4j makes it easier to get started for everyone who knows SQL from working with relational databases. Developers as well as operations and business users can run ad-hoc queries on the graph for a variety of use cases. Cypher draws its inspiration from a variety of sources: SQL, SparQL, ASCII-Art, and functional programming. The core concept is that the user describes the patterns to be matched in the graph and supplies starting points. The database engine then efficiently matches the given patterns across the graph, enabling users to define sophisticated queries like "find me all the customers who have friends who have recently bought similar products." Like other query languages, it supports filtering, grouping, and paging. Cypher allows easy creation, deletion, update, and graph construction.

The Cypher statement in Example 7-2 shows a typical use case. It starts by looking up a customer from an index and then following relationships via his orders to the products he ordered. Filtering out older orders, the query then calculates the top 20 largest volumes he purchased by product.

Example 7-2. Sample Cypher statement

```
START    customer=node:Customer(email = "dave@dmband.com")
MATCH    customer-[:ORDERED]->order-[item:LINEITEM]->product
WHERE    order.date > 20120101
RETURN   product.name, sum(item.amount) AS product
ORDER BY products DESC
LIMIT    20
```

Being written in Java, Neo4j is easily embeddable in any Java application which refers to single-instance deployments. However, many deployments of Neo4j use the standalone Neo4j server, which offers a convenient HTTP API for easy interaction as well as a comprehensive web interface for administration, exploration, visualization, and monitoring purposes. The Neo4j server is a simple download, and can be uncompressed and started directly.

It is possible to run the Neo4j server on top of an embedded database, which allows easy access to the web interface for inspection and monitoring (Figure 7-2).

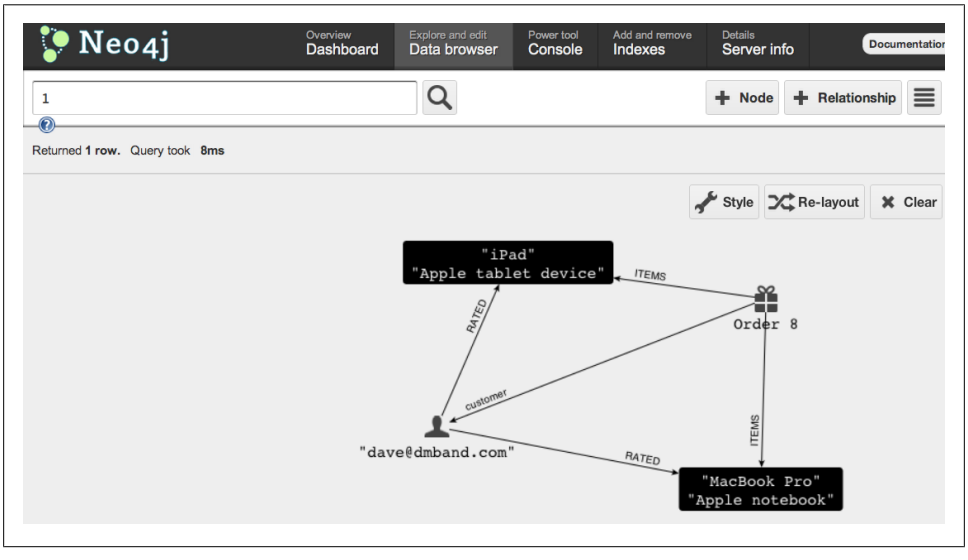


Figure 7-2. Neo4j server web interface

In the web interface, you can see statistics about your database. In the data browser, you can find nodes by ID, with index lookups, and with cypher queries (click the little blue question mark for syntax help), and switch to the graph visualizer with the right-hand button to explore your graph visually (as shown in Figure 7-2). The console allows you to enter Cypher statements directly or even issue HTTP requests. Server Info lists JMX beans, which, especially in the Enterprise edition, come with much more information.

As an open source product, Neo4j has a very rich and active ecosystem of contributors, community members, and users. Neo Technology, the company sponsoring the development of Neo4j, makes sure that the open source licensing (GPL) for the community edition, as well as the professional support for the enterprise editions, promote the continuous development of the product.

To access Neo4j, you have a variety of drivers available, most of them being maintained by the community. There are libraries for many programming languages for both the embedded and the server deployment mode. Some are maintained by the Neo4j team, Spring Data Neo4j being one of them.

Spring Data Neo4j Overview

Spring Data Neo4j was the original Spring Data project initiated by Rod Johnson and Emil Eifrem. It was developed in close collaboration with VMware and Neo Technology and offers Spring developers an easy and familiar way to interact with Neo4j. It intends to leverage the well-known annotation-based programming models with a tight integration in the Spring Framework ecosystem. As part of the Spring Data project, Spring Data Neo4j integrates both Spring Data Commons repositories (see Chapter 2) as well as other common infrastructures.

As in JPA, a few annotations on POJO (plain old Java object) entities and their fields provide the necessary metainformation for Spring Data Neo4j to map Java objects into graph elements. There are annotations for entities being backed by nodes (`@NodeEntity`) or relationships (`@RelationshipEntity`). Field annotations declare relationships to other entities (`@RelatedTo`), custom conversions, automatic indexing (`@Indexed`), or computed/derived values (`@Query`). Spring Data Neo4j allows us to store the type information (hierarchy) of the entities for performing advanced operations and type conversions. See Example 7-3.

Example 7-3. An annotated domain class

```
@NodeEntity
public class Customer {
  @GraphId Long id;

  String firstName, lastName;

  @Indexed(unique = true)
  String emailAddress;

  @RelatedTo(type = "ADDRESS")
  Set<Address> addresses = new HashSet<Address>();
}
```

The core infrastructure of Spring Data Neo4j is the `Neo4jTemplate`, which offers (similar to other template implementations) a variety of lower-level functionality that encapsulates the Neo4j API to support mapped domain objects. The Spring Data Neo4j infrastructure and the repository implementation uses the `Neo4jTemplate` for its operations. Like the other Spring Data projects, Spring Data Neo4j is configured via two XML namespace elements—for general setup and repository configuration.

To tailor Neo4j to individual use cases, Spring Data Neo4j supports both the embedded mode of Neo4j as well as the server deployment, where the latter is accessed via Neo4j's Java-REST binding. Two different mapping modes support the custom needs of developers. In the simple mapping mode, the graph data is copied into domain objects, being detached from the graph. The more advanced mapping mode leverages AspectJ to provide a live, connected representation of the graph elements bound to the domain objects.

Modeling the Domain as a Graph

The domain model described in Chapter 1 is already a good fit for a graph database like Neo4j (see Figure 7-3). To allow some more advanced graph operations, we're going to normalize it further and add some additional relationships to enrich the model.

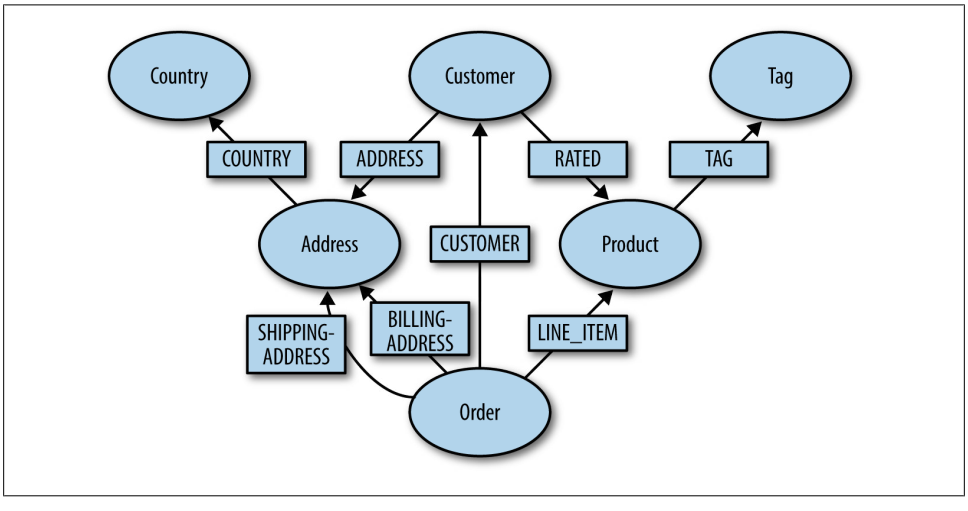


Figure 7-3. Domain model as a graph

The code samples listed here are not complete but contain the necessary information for understanding the mapping concepts. See the Neo4j project in the sample source-repository for a more complete picture.

In Example 7-4, the `AbstractEntity` as a superclass was kept with the same `id` field (which got a `@GraphId` annotation and `equals(…)` and `hashCode()` methods, as previously discussed). Annotating the `id` is required in the simple mapping mode, as it is the only way to keep the node or relationship `id` stored in the entity. Entities can be be loaded by their `id` with `Neo4jTemplate.findOne()`, and a similar method exists in the `GraphRepository`.

Example 7-4. Base domain class

```
public abstract class AbstractEntity {

  @GraphId
  private Long id;
}
```

The simplest mapped class is just marked with `@NodeEntity` to make it known to Spring Data Neo4j's mapping infrastructure. It can contain any number of primitive fields, which will be treated as node properties. Primitive types are mapped directly. Types

not supported by Neo4j can be converted to equivalent primitive representations by supplied Spring converters. Converters for `Enum` and `Date` fields come with the library.

In `Country`, both fields are just simple strings, as shown in Example 7-5. The `code` field represents a unique "business" key and is marked as `@Indexed(unique=true)` which causes the built-in facilities for unique indexes to be used; these are exposed via `Neo4jTemplate.getOrCreateNode()`. There are several methods in the `Neo4jTemplate` to access the Neo4j indexes; we can find entities by their indexed keys with `Neo4jTemplate.lookup()`.

Example 7-5. Country as a simple entity

```
@NodeEntity
public class Country extends AbstractEntity {

  @Indexed(unique=true)
  String code;
  String name;
}
```

Customers are stored as nodes; their unique key is the `emailAddress`. Here we meet the first references to other objects (in this case, `Address`), which are represented as relationships in the graph. So fields of single references or collections of references always cause relationships to be created when updated, or navigated when accessed.

As shown in Example 7-6, reference fields can be annotated with `@RelatedTo`, to document the fact that they are reference fields or set custom attributes like the relationship type (in this case, `"ADDRESS"`). If we do not provide the type, it defaults to the field name. The relationship points by default to the referred object (`Direction.OUTGOING`), the opposite direction can be specified in the annotation; this is especially important for bidirectional references, which should be mapped to just a single relationship.

Example 7-6. Customer has relationships to his addresses

```
@NodeEntity
public class Customer extends AbstractEntity {

  private String firstName, lastName;

  @Indexed(unique = true)
  private String emailAddress;

  @RelatedTo(type = "ADDRESS")
  private Set<Address> addresses = new HashSet<Address>();
}
```

The `Address` is pretty simple again. Example 7-7 shows how the `country` reference field doesn't have to be annotated—it just uses the field name as the relationship type for the outgoing relationship. The customers connected to this address are not represented in the mapping because they are not necessary for our use case.

Example 7-7. Address connected to country

```
@NodeEntity
public class Address extends AbstractEntity {

  private String street, city;
  private Country country;
}
```

The `Product` has a unique name and shows the use of a nonprimitive field; the price will be converted to a primitive representation by Springs' converter facilities. You can register your own converters for custom types (e.g., value objects) in your application context.

The description field will be indexed by an index that allows full-text search. We have to name the index explicitly, as it uses a different configuration than the default, exact index. You can then find the products by calling, for instance, `neo4jTemplate.lookup("search","description:Mac*")`, which takes a Lucene query string.

To enable interesting graph operations, we added a `Tag` entity and relate to it from the `Product`. These tags can be used to find similar products, provide recommendations, or analyze buying behavior.

To handle dynamic attributes of an entity (a map of arbitrary key/values), there is a special support class in Spring Data Neo4j. We decided against handling maps directly because they come with a lot of additional semantics that don't fit in the context. Currently, `DynamicProperties` are converted into properties of the node with prefixed names for separation. (See Example 7-8.)

Example 7-8. Tagged product with custom dynamic attributes

```
@NodeEntity
public class Product extends AbstractEntity {

  @Indexed(unique = true)
  private String name;
  @Indexed(indexType = IndexType.FULLTEXT, indexName = "search")
  private String description;
  private BigDecimal price;

  @RelatedTo
  private Set<Tag> tags = new HashSet<Tag> ();
  private DynamicProperties attributes = new PrefixedDynamicProperties("attributes");
}
```

The only unusual thing about the `Tag` is the `Object value` property. This property is converted according to the runtime value into a primitive value that can be stored by Neo4j. The `@GraphProperty` annotation, as shown in Example 7-9, allows some customization of the storage (e.g., the used property name or a specification of the primitive target type in the graph).

Example 7-9. A simple Tag

```
@NodeEntity
public class Tag extends AbstractEntity {

  @Indexed(unique = true)
  String name;

  @GraphProperty
  Object value;
}
```

The first `@RelationshipEntity` we encounter is something new that didn't exist in the original domain model but which is nonetheless well known from any website. To allow for some more interesting graph operations we add a `Rating` relationship between a `Customer` and a `Product`. This entity is annotated with `@RelationshipEntity` to mark it as such. Besides two simple fields holding the rating `stars` and a `comment`, we can see that it contains fields for the actual start and end of the relationship, which are annotated appropriately (Example 7-10).

Example 7-10. A Rating between Customer and Product

```
@RelationshipEntity(type = "RATED")
public class Rating extends AbstractEntity {
    @StartNode Customer customer;
    @EndNode Product product;
    int stars;
    String comment;
}
```

Relationship entities can be created as normal POJO classes, supplied with their start and endpoints, and saved via `Neo4jTemplate.save()`. In Example 7-11, we show with the `Order` how these entities can be retrieved as part of the mapping. In the more in-depth discussion of graph operations—see "Leverage Similar Interests (Collaborative Filtering)" on page 121—we'll see how to leverage those relationships in Cypher queries with `Neo4jTemplate.query` or repository finder methods.

The `Order` is the most connected entity so far; it sits in the middle of our domain. In Example 7-11, the relationship to the `Customer` shows the inverse `Direction.INCOMING` for a bidirectional reference that shares the same relationship.

The easiest way to model the different types of addresses (shipping and billing) is to use different relationship types—in this case, we just rely on the different field names. Please note that a single address object/node can be used in multiple places for example, as both the shipping and billing address of a single customer, or even across customers (e.g., for a family). In practice, a graph is often much more normalized than a relational database, and the removal of duplication actually offers multiple benefits both in terms of storage and the ability to run more interesting queries.

Example 7-11. Order, the centerpiece of the domain

```
@NodeEntity
public class Order extends AbstractEntity {

  @RelatedTo(type = "ORDERED", direction = Direction.INCOMING)
  private Customer customer;

  @RelatedTo
  private Address billingAddress;

  @RelatedTo
  private Address shippingAddress;

  @Fetch
  @RelatedToVia
  private Set<LineItem> lineItems = new HashSet<LineItem>();
}
```

The `LineItem`s are not modeled as nodes but rather as relationships between `Order` and `Product`. A `LineItem` has no identity of its own and just exists as long as both its endpoints exist, which it refers to via its `order` and `product` fields. In this model, `LineItem` only contains the `quantity` attribute, but in other use cases, it can also contain different attributes.

The interesting pieces in `Order` and `LineItem` are the `@RelatedToVia` annotation and `@Fetch`, which is discussed shortly. The annotation on the `lineItems` field is similar to `@RelatedTo` in that it applies only to references to relationship entities. It is possible to specify a custom relationship type or direction. The type would override the one provided in the `@RelationshipEntity` (see Example 7-12).

Example 7-12. A LineItem is just a relationship

```
@RelationshipEntity(type = "ITEMS")
public class LineItem extends AbstractEntity {

  @StartNode private Order order;

  @Fetch
  @EndNode
  private Product product;
  private int amount;
}
```

This takes us to one important aspect of object-graph mapping: fetch declarations. As we know from JPA, this can be tricky. For now we've kept things simple in Spring Data Neo4j by not fetching related entities by default.

Because the simple mapping mode needs to copy data out of the graph into objects, it must be careful about the fetch depth; otherwise you can easily end up with the whole graph pulled into memory, as graph structures are often cyclic. That's why the default strategy is to load related entities only in a shallow way. The `@Fetch` annotation is used

to declare fields to be loaded eagerly and fully. We can load them after the fact by `template.fetch(entity.field)`. This applies both to single relationships (one-to-one) and multi-relationship fields (one-to-many).

In the `Order`, the `LineItems` are fetched by default, becuse they are important in most cases when an order is loaded. For the `LineItem` itself, the `Product` is eagerly fetched so it is directly available. Depending on your use case, you would model it differently.

Now that we have created the domain classes, it's time to store their data in the graph.

Persisting Domain Objects with Spring Data Neo4j

Before we can start storing domain objects in the graph, we should set up the project. In addition to your usual Spring dependencies, you need either `org.springframework.data:spring-data-neo4j:2.1.0.RELEASE` (for simple mapping) or `org.springframework.data:spring-data-neo4j-aspects:2.1.0.RELEASE` (for advanced AspectJ-based mapping (see "Advanced Mapping Mode" on page 123) as a dependency. Neo4j is pulled in automatically (for simplicity, assuming the embedded Neo4j deployment).

The minimal Spring configuration is a single namespace config that also sets up the graph database (Example 7-13).

Example 7-13. Spring configuration setup

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:neo4j="http://www.springframework.org/schema/data/neo4j"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/data/neo4j
        http://www.springframework.org/schema/data/neo4j/spring-neo4j.xsd">


    <neo4j:config storeDirectory="target/graph.db" />
    <neo4j:repositories base-package="com.oreilly.springdata.neo4j" />

</beans>
```

As shown in Example 7-14, we can also pass a `graphDatabaseService` instance to `neo4j:config`, in order to configure the graph database in terms of caching, memory usage, or upgrade policies. This even allows you to use an in-memory `ImpermanentGraphDatabase` for testing.

Example 7-14. Passing a graphDatabaseService to the configuration

```
<neo4j:config graphDatabaseService="graphDatabaseService" />

<bean id="graphDatabaseService" class="org.neo4j.test.ImpermanentGraphDatabase" />

<!-- or -->
```

```
<bean id="graphDatabaseService" class="org.neo4j.kernel.EmbeddedGraphDatabase"
        destroy-method="shutdown">
    <constructor-arg value="target/graph.db" />
    <constructor-arg> <!-- passing configuration properties -->
        <map>
            <entry key="allow_store_upgrade" value="true" />
        </map>
    </constructor-arg>
</bean>
```

After defining the domain objects and the setup, we can pretty easily generate the sample dataset that will be used to illustrate some use cases (see Example 7-15 and Figure 7-4). Both the domain classes, as well as the dataset generation and integration tests documenting the use cases, can be found in the GitHub repository for the book (see "The Sample Code" on page 6 for details). To import the data, we can simply populate domain classes and use `template.save(entity)`, which either merges the entity with the existing element in the graph or creates a new one. That depends on mapped IDs and possibly unique field declarations, which would be used to identify existing entities in the graph with which we're merging.

Example 7-15. Populating the graph with the sample dataset

```
Customer dave = template.save(new Customer("Dave", "Matthews", "dave@dmband.com"));
template.save(new Customer("Carter","Beauford","carter@dmband.com"));
template.save(new Customer("Boyd","Tinsley","boyd@dmband.com"));

Country usa = template.save(new Country("US", "United States"));
template.save(new Address("27 Broadway","New York",usa));

Product iPad = template.save(new Product("iPad", "Apple tablet device").withPrice(499));
Product mbp = template.save(new Product("MacBook Pro", "Apple notebook").withPrice(1299));

template.save(new Order(dave).withItem(iPad,2).withItem(mbp,1));
```

The entities shown here use some convenience methods for construction to provide a more readable setup (Figure 7-4).

Neo4jTemplate

The Neo4jTemplate is like other Spring templates: a convenience API over a lower-level one, in this case the Neo4j API. It adds the usual benefits, like transaction handling and exception translation, but more importantly, automatic mapping from and to domain entities. The Neo4jTemplate is used in the other infrastructural parts of Spring Data Neo4j. Set it up by adding the `<neo4j:config/>` declaration to your application context or by creating a new instance, which is passed a Neo4j `GraphDatabaseService` (which is available as a Spring bean and can be injected into your code if you want to access the Neo4j API directly).

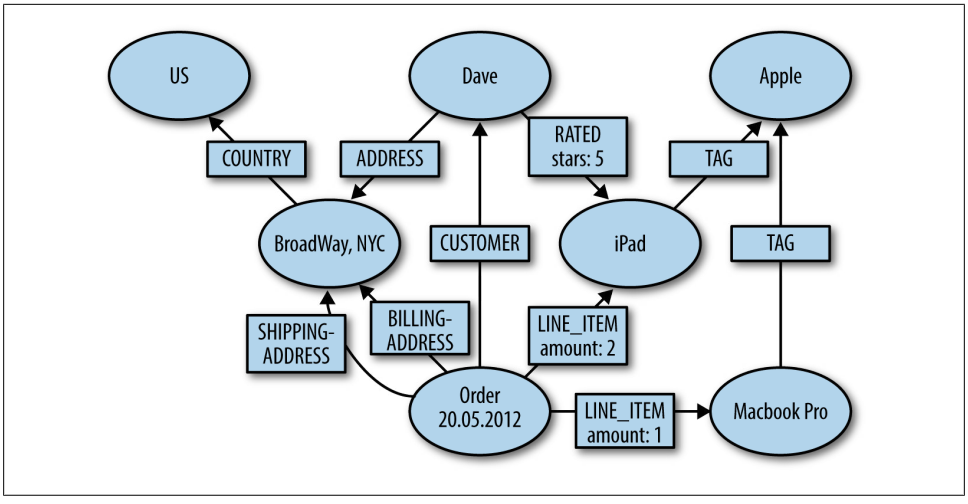


Figure 7-4. Graph of imported domain data

The operations for creating entities, nodes, and relationships and finding or removing them by `id` comprise the basics (`save()`, `getOrCreateNode()`, `findOne()`, `getNode()`, `getRelationshipsBetween()`, etc.). Most of the other mechanisms deal with more advanced ways to look up interesting things in the graph—by issuing index queries with `lookup`, executing Cypher statements with `query()`, or running a traversal with `traverse()`. The `Neo4jTemplate` offers methods to convert nodes into entities with `load()`, or one entity into a different type with `projectTo()` (see "Multiple Roles for a Single Node" on page 119). Lazily loaded entities can be loaded fully via `fetch()`.

You can achieve most of what you want to do with Spring Data Neo4j with the `Neo4jTemplate` alone, but the repository support adds a much more convenient way to perform many operations.

Combining Graph and Repository Power

With all that set up, we can now look into how repositories integrate with Spring Data Neo4j and how they are used in a "graphy" way.

Spring Data Commons repositories (see Chapter 2) make it easy to keep persistence access related code (or rather noncode) in one place and allow us to write as little of it as possible to satisfy the specific use cases. In Spring Data Neo4j, repositories are derived from the `GraphRepository<T>` base interface, which already combines some of the usually needed functionality: CRUD operations, and index and traversal functions. The basic setup for repositories is just another line of the namespace configuration, as shown in Example 7-16. Each domain class will be bound to an individual, concrete repository interface (see Example 7-17).

Example 7-16. Basic repository configuration setup

```
<neo4j:repositories base-package="com.oreilly.springdata.neo4j" />
```

Example 7-17. Basic repository interface declaration

```
import org.springframework.data.neo4j.repository.GraphRepository;

public interface CustomerRepository extends GraphRepository<Customer> {

  Customer findByEmailAddress(String emailAddress);
}
```

Spring Data Neo4j repositories provide support for `@Query`-annotated and derived finder methods, which are projected to Cypher statements. To understand how this mapping works, you need to be aware of the expressive syntax of Cypher, which is explained in the next sidebar, "Cypher Query Language".

Cypher Query Language

Neo4j comes with a clean, object-oriented Java API and enjoys many JVM (Java virtual machine) language bindings as well as a plethora of drivers for the Neo4j server. But often data operations are better expressed declaratively by asking "what" than by specifying "how."

That's why the Cypher query language was developed. It builds upon matching patterns in a graph that are bound to specified nodes and relationships and allows further filtering and paging of the results. Cypher has data manipulation features that allow us to modify the graph. Cypher query parts can be chained (piped) to enable more advanced and powerful graph operations.

Each Cypher query can consist of several parts:

START
: Defines identifiers, binding nodes, and relationships either by index or ID lookup.
`START user=node:customers(name="dave@...")`

MATCH
: Uses ASCII-ART descriptions for patterns to find in the graph. Patterns are bound to identifiers and define new identifiers. Each of the subgraphs found during the query execution spawns an individual result.

```
MATCH user-[rating:RATED]->product
```

WHERE
: Filters the result using boolean expressions, and uses dot notation for accessing properties, functions, collection predicates and functions, arithmetic operators, etc.

```
WHERE user.name = "Dave" AND ANY(color in product.colors : color = 'red')
OR rating.stars > 3
```

SKIP LIMIT
: Paginates the results with offsets and sizes.

```
SKIP 20 LIMIT 10
```

RETURN
: Declares what to return from the query. If aggregation functions are used, all non-aggregated values will be used as grouping values.

```
return user.name, AVG(rating.stars) AS WEIGHT, product
```

ORDER BY
: Orders by properties or any other expression. `ORDER BY user.name ASC, count(*) DESC`

UPDATES
: There is more to Cypher. With `CREATE [UNIQUE], SET, DELETE`, the graph can be modified on the fly. `WITH` and `FOREACH` allow for more advanced query structures.

PARAMETERS
: Cypher can be passed in a map of parameters which can be referenced by key (or position). `start n=node({nodeId}) where n.name=~{0} return n`

The results returned by Cypher are inherently tabular, much like `JDBC ResultSets`. The column names serve as row-value keys.

There is a Java DSL for Cypher that, instead of using semantic-free strings for queries, offers a type-safe API to build up Cypher queries. It allows us to optionally leverage Querydsl (see Chapter 3) to build expressions for filters and index queries out of generated domain object literals. With an existing JDBC driver, cypher queries can be easily integrated into existing Java (Spring) applications and other JDBC tools.

Basic Graph Repository Operations

The basic operations provided by the repositories mimic those offered by the `Neo4jTemplate`, only bound to the declared repository domain class. So `findOne(…)`, `save(…)`, `delete(…)`, `findAll(…)`, and so on, take and return instances of the domain class.

Spring Data Neo4j stores the type (hierarchy) information of the mapped entities in the graph. It uses one of several strategies for this purpose, defaulting to an index-based storage. This type information is used for all repository and template methods that operate on all instances of a type and for verification of requested types versus stored types.

The updating repository methods are transactional by default, so there is no need to declare a transaction around them. For domain use cases, however, it is sensible to do so anyway, as usually more than one database operation is encapsulated by a business transaction. (This uses the Neo4j supplied support for `JtaTransactionManager`)

For index operations, specific methods like `findAllByPropertyValue()`, `findAllByQuery()`, and `findAllByRange()` exist in the `IndexRepository` and are mapped directly

to the underlying index infrastructure of Neo4j, but take the repository domain class and existing index-related annotations into account. Similar methods are exposed in the `TraversalRepository` whose `findAllByTraversal()` method allows direct access to the powerful graph traversal mechanisms of Neo4j. Other provided repository interfaces offer methods for spatial queries or the Cypher-DSL integration.

Derived and Annotated Finder Methods

Besides the previously discussed basic operations, Spring Data Neo4j repositories support custom finder methods by leveraging the Cypher query language. For both annotated and derived finder methods, additional `Pageable` and `Sort` method parameters are taken into account during query execution. They are converted into appropriate `ORDER BY`, `SKIP`, and `LIMIT` declarations.

Annotated finder methods

Finders can use Cypher directly if we add a `@Query` annotation that contains the query string, as shown in Example 7-18. The method arguments are passed as parameters to the Cypher query, either via their parameter position or named according to their `@Parameter` annotation, so you can use `{index}` or `{name}` in the query string.

Example 7-18. An annotated cypher query on a repository query method

```
public interface OrderRepository extends GraphRepository<Order> {

  @Query(" START c=node({0}) " +
         " MATCH c-[:ORDERED]->order-[item:LINE_ITEM]->product " +
         "  WITH order, SUM (product.price * item.amount) AS value " +
         " WHERE value > {orderValue} " +
         "RETURN order")
  Collection<Order> findOrdersWithMinimumValue(Customer customer,
                                @Parameter("orderValue") int value);
}
```

Result handling

The return types of finder methods can be either an `Iterable<T>`, in which case the evaluation of the query happens lazily, or any of these interfaces: `Collection<T>`, `List<T>`, `Set<T>`, `Page<T>`. `T` is the result type of the query, which can be either a mapped domain entity (when returning nodes or relationships) or a primitive type. There is support for an interface-based simple mapping of query results. For mapping the results, we have to create an interface annotated with `@MapResult`. In the interface we declare methods for retrieving each column-value. We annotate the methods individually with `@ResultColumn("columnName")`. See Example 7-19.

Example 7-19. Defining a MapResult and using it in an interface method

```
@MapResult
interface RatedProduct {
```

```
  @ResultColumn("product")
  Product getProduct();

  @ResultColumn("rating")
  Float getRating();

  @ResultColumn("count")
  int getCount();
}

public interface ProductRepository extends GraphRepository<Product> {

  @Query(" START tag=node({0}) " +
         " MATCH tag-[:TAG]->product<-[rating:RATED]-() " +
          "RETURN product, avg(rating.stars) AS rating, count(*) as count " +
          " ORDER BY rating DESC")
  Page<RatedProduct> getTopRatedProductsForTag(Tag tag, Pageable page);
}
```

To avoid the proliferation of query methods for different granularities, result types, and container classes, Spring Data Neo4j provides a small fluent API for result handling. The API covers automatic and programmatic value conversion. The core of the result handling API centers on converting an iterable result into different types using a configured or given `ResultConverter`, deciding on the granularity of the result size and optionally on the type of the target container. See Example 7-20.

Example 7-20. Result handling API

```
public interface ProductRepository extends GraphRepository<Product> {

  Result<Map<String,Object>> findByName(String name);
}

Result<Map<String,Object>> result = repository.findByName("mac");

// return a single node (or null if nothing found)
Node n = result.to(Node.class).singleOrNull();
Page<Product> page = result.to(Product.class).as(Page.class);

Iterable<String> names = result.to(String.class,
  new ResultConverter<Map<String, Object>, String>() {
    public String convert(Map<String, Object> row) {
       return (String) ((Node) row.get("name")).getProperty("name");
    }
  });
```

Derived finder methods

As described in Chapter 2, the derived finder methods (see "Property expressions" on page 17) are a real differentiator. They leverage the existing mapping information about the targeted domain entity and an intelligent parsing of the finder method name to generate a query that fetches the information needed.

Derived finder methods—like `ProductRepository.findByNameAndColorAndTagName(name, color, tagName)`—start with `find(By)` or `get(By)` and then contain a succession of property expressions. Each of the property expressions either points to a property name of the current type or to another, related domain entity type and one of its properties. These properties must exist on the entity. If that is not the case, the repository creation fails early during `ApplicationContext` startup.

For all valid finder methods, the repository constructs an appropriate query by using the mapping information about domain entities. Many aspects—like in-graph type representation, indexing information, field types, relationship types, and directions—are taken into account during the query construction. This is also the point at which appropriate escaping takes place.

Thus, Example 7-20 would be converted to the query shown in Example 7-21.

Example 7-21. Derived query generation

```
@NodeEntity
class Product {

  @Indexed
  String name;
  int price;

  @RelatedTo(type = "TAG")
  Set<Tag> tags;
}

@NodeEntity
class Tag {

  @Indexed
  String name;
}

public interface ProductRepository extends GraphRepository<Product> {

  List<Product> findByNameAndPriceGreaterThanAndTagsName(String name, int price,
  String tagName);
}

// Generated query
 START product = node:Product(name = {0}), productTags = node:Tag(name = {3})
 MATCH product-[:TAG]->productTags
 WHERE product.price > {1}
RETURN product
```

This example demonstrates the use of index lookups for indexed attributes and the simple property comparison. If the method name refers to properties on other, related entities, then the query builder examines those entities for inclusion in the generated query. The builder also adds the direction and type of the relationship to that entity. If there are more properties further along the path, the same action is repeated.

Supported keywords for the property comparison are:

- Arithmetic comparisons like `GreaterThan`, `Equals`, or `NotEquals`.
- `IsNull` and `IsNotNull` check for `null` (or nonexistent) values.
- `Contains`, `StartsWith`, `EndsWith` and `Like` are used for string comparison.
- The `Not` prefix can be used to negate an expression.
- `Regexp` for matching regular expressions.

For many of the typical query use cases, it is easy enough to just code a derived finder declaration in the repository interface and use it. Only for more involved queries is an annotated query, traversal description, or manual traversing by following relationships necessary.

Advanced Graph Use Cases in the Example Domain

Besides the ease of mapping real-world, connected data into the graph, using the graph data model allows you to work with your data in interesting ways. By focusing on the value of relationships in your domain, you can find new insights and answers that are waiting to be revealed in the connections.

Multiple Roles for a Single Node

Due to the schema-free nature of Neo4j, a single node or relationship is not limited to be mapped to a single domain class. Sometimes it is sensible to structure your domain classes into smaller concepts/roles that are valid for a limited scope/context.

For example, an `Order` is used differently in different stages of its life cycle. Depending on the current state, it is either a shopping cart, a customer order, a dispatch note, or a return order. Each of those states is associated with different attributes, constraints, and operations. Usually, this would have been modeled either in different entities stored in separate tables or in a single `Order` class stored in a very large and sparse table row. With the schemaless nature of the graph database, the order will be stored in a node but only contains the state (and relationships) that are needed in the current state (and those still needed from past states). Usually, it gains attributes and relationships during its life, and gets simplified and locked down only when being retired.

Spring Data Neo4j allows us to model such entities with different classes, each of which covers one period of the life cycle. Those entities share a few attributes; each has some unique ones. All entities are mapped to the same node, and depending on the type provided at load time with `template.findOne(id,type)`, or at runtime with `template.projectTo(object, type)`, it can be used differently in different contexts. When the projected entity is stored, only its current attributes (and relationships) are updated; the other existing ones are left alone.

Product Categories and Tags as Examples for In-Graph Indexes

For handling larger product catalogs and ease of exploration, it is important to be able to put products into categories. A naive approach that uses a single category attribute with just one value per product falls short in terms of long-term usability. In a graph, multiple connections to category nodes per entity are quite natural. Adding a tree of categories, where each has relationships to its children and each product has relationships to the categories it belongs to, is really simple. Typical use cases are:

- Navigation of the category tree
- Listing of all products in a category subtree
- Listing similar products in the same category
- Finding implicit/non-obvious relationships between product categories (e.g., baby care products and lifestyle gadgets for young parents)

The same goes for tags, which are less restrictive than categories and often form a natural graph, with all the entities related to tags instead of a hierarchical tree like categories. In a graph database, both multiple categories as well as tags form implicit secondary indexing structures that allow navigational access to the stored entities in many different ways. There can be other secondary indexes (e.g., geoinformation, time-related indices, or other interesting dimensions). See Example 7-22.

Example 7-22. Product categories and tags

```
@NodeEntity
public class Category extends AbstractEntity {
  @Indexed(unique = true) String name;
  @Fetch // loads all children eagerly (cascading!)
  @RelatedTo(type="SUB_CAT")
  Set<Category> children = new HashSet<Category>();

  public void addChild(Category cat) {
    this.children.add(cat);
  }
}

@NodeEntity
public class Product extends AbstractEntity {
  @RelatedTo(type="CATEGORY")
  Set<Category> categories = new HashSet<Category>();

  public void addCategory(Category cat) {
    this.categories.add(cat);
  }
}

public interface ProductRepository extends GraphRepository<Product> {
  @Query("START cat=node:Category(name={0}) "+
         "MATCH cat-[SUB_CAT*0..5]-leaf<-[:CATEGORY]-product "+
         "RETURN distinct product")
```

```
  List<Product> findByCategory(String category);
}
```

The `Category` forms a nested composite structure with parent-child relationships. Each category has a unique name and a set of children. The category objects are used for creating the structure and relating products to categories. For leveraging the connectedness of the products, a custom (annotated) query navigates from a start (or root) category, via the next zero through five relationships, to the products connected to this subtree. All attached products are returned in a list.

Leverage Similar Interests (Collaborative Filtering)

`Collaborative filtering`, demonstrated in Example 7-23, relies on the assumption that we can find other "people" who are very similar/comparable to the current user in their interests or behavior. Which criteria are actually used for similarity—search/buying history, reviews, or others—is domain-specific. The more information the algorithm gets, the better the results.

In the next step, the products that those similar people also bought or liked are taken into consideration (measured by the number of their mentions and/or their rating scores) optionally excluding the items that the user has already bought, owns, or is not interested in.

Example 7-23. Collaborative filtering

```
public interface ProductRepository extends GraphRepository<Product> {
    @Query("START cust=node({0}) " +
           " MATCH cust-[r1:RATED]->product<-[r2:RATED]-people " +
           "           -[:ORDERED]->order-[:ITEMS]->suggestion " +
           " WHERE abs(r1.stars - r2.stars) <= 2 " +
           " RETURN suggestion, count(*) as score" +
           " ORDER BY score DESC")
    List<Suggestion> recommendItems(Customer customer);

    @MapResult
    interface Suggestion {
        @ResultColumn("suggestion") Product getProduct();
        @ResultColumn("score") Integer getScore();
    }
}
```

Recommendations

Generally in all domains, but particularly in the ecommerce domain, making recommendations of interesting products for customers is key to leveraging the collected information on product reviews and buying behavior. Obviously, we can derive recommendations from explicit customer reviews, especially if there is too little actual buying history or no connected user account. For the initial suggestion, a simple

ordering of listed products by number and review rating (or more advanced scoring mechanisms) is often sufficient.

For more advanced recommendations, we use algorithms that take multiple input data vectors into account (e.g., ratings, buying history, demographics, ad exposure, and geo-information).

The query in Example 7-24 looks up a product and all the ratings by any customer and returns a single page of top-rated products (depending on the average rating).

Example 7-24. Simple recommendation

```
public interface ProductRepository extends GraphRepository<Product> {
  @Query("START product=node:product_search({0}) "+
         "MATCH product<-[r:RATED]-customer "+
         "RETURN product ORDER BY avg(r.stars) DESC"
  Page<Product> listProductsRanked(String description, Pageable page);

}
```

Transactions, Entity Life Cycle, and Fetch Strategies

With Neo4j being a fully transactional database, Spring Data Neo4j participates in (declarative) Spring transaction management, and builds upon transaction managers provided by Neo4j that are compatible with the Spring `JtaTransactionManager`. The transaction-manager bean named `neo4jTransactionManager` (aliased to `transactionManager`) is created in the `<neo4j:config />` element. As transaction management is configured by default, `@Transactional` annotations are all that's needed to define transactional scopes. Transactions are needed for all write operations to the graph database, but reads don't need transactions. It is possible to nest transactions, but nested transactions will just participate in the running parent transaction (like `REQUIRED`).

Spring Data Neo4j, as well as Neo4j itself, can integrate with external XA transaction managers; the Neo4j manual describes the details.

For the simple mapping mode, the life cycle is straightforward: a new entity is just a POJO instance until it has been stored to the graph, in which case it will keep the internal `id` of the element (node or relationship) in the `@GraphId` annotated field for later reattachment or merging. Without the `id` set, it will be handled as a new entity and trigger the creation of a new graph element when saved.

Whenever entities are fetched in simple mapping mode from the graph, they are automatically detached. The data is copied out of the graph and stored in the domain object instances. An important aspect of using the simple mapping mode is the `fetch depth`. As a precaution, the transaction fetches only the direct properties of an entity and doesn't follow relationships by default when loading data.

To achieve a deeper fetch graph, we need to supply a `@Fetch` annotation on the fields that should be eagerly fetched. For entities and fields not already fetched, the `template.fetch(…)` method will load the data from the graph and update them in place.

Advanced Mapping Mode

Spring Data Neo4j also offers a more advanced mapping mode. Its main difference from the simple mapping mode is that it offers a live view of the graph projected into the domain objects. So each field access will be intercepted and routed to the appropriate properties or relationships (for `@RelatedTo[Via]` fields). This interception uses AspectJ under the hood to work its magic.

We can enable the advanced mapping mode by adding the `org.springframework.data:spring-data-neo4j-aspects` dependency and configuring either a AspectJ build plug-in or load-time-weaving activation (Example 7-25).

Example 7-25. Spring Data Neo4j advanced mapping setup

```
<properties>
  <aspectj.version>1.6.12</aspectj.version>
</properties>

<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-neo4j-aspects</artifactId>
  <version>${spring-data-neo4j.version}</version>
</dependency>
<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjrt</artifactId>
  <version>${aspectj.version}</version>
</dependency>

....
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>aspectj-maven-plugin</artifactId>
  <version>1.2</version>
  <dependencies>
    <dependency>
      <groupId>org.aspectj</groupId>
      <artifactId>aspectjrt</artifactId>
      <version>${aspectj.version}</version>
    </dependency>
    <dependency>
      <groupId>org.aspectj</groupId>
      <artifactId>aspectjtools</artifactId>
      <version>${aspectj.version}</version>
    </dependency>
  </dependencies>
  <executions>
    <execution>
```

```
      <goals>
        <goal>compile</goal>
        <goal>test-compile</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <outxml>true</outxml>
    <aspectLibraries>
      <aspectLibrary>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aspects</artifactId>
      </aspectLibrary>
      <aspectLibrary>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-neo4j-aspects</artifactId>
      </aspectLibrary>
    </aspectLibraries>
    <source>1.6</source>
    <target>1.6</target>
  </configuration>
</plugin>
```

Fields are automatically read from the graph at any time, but for immediate write-through the operation must happen inside of a transaction. Because objects can be modified outside of a transaction, a life cycle of attached/detached objects has been established. Objects loaded from the graph or just `save`d inside a transaction are *attached*; if an object is modified outside of a transaction or newly created, it is `detached`. Changes to detached objects are stored in the object itself, and will only be reflected in the graph with the next `save` operation, causing the entity to become attached again.

This live view of the graph database allows for faster operation as well as "direct" manipulation of the graph. Changes will be immediately visible to other graph operations like traversals, Cypher queries, or Neo4j Core API methods. Because reads *always* happen against the live graph, all changes by other committed transactions are immediately visible. Due to the immediate live reads from the graph database, the advanced mapping mode has no need of fetch handling and the `@Fetch` annotation.

Working with Neo4j Server

We've already mentioned that Neo4j comes in two flavors. You can easily use the high-performance, embeddable Java database with any JVM language, preferably with that language's individual idiomatic APIs/drivers. Integrating the embedded database is as simple as adding the Neo4j libraries to your dependencies.

The other deployment option is Neo4j server. The Neo4j server module is a simple download or operating system package that is intended to be run as an independent service. Access to the server is provided via a web interface for monitoring, operations,

and visualizations (refer back to Example 7-1). A comprehensive REST API offers programmatic access to the database functionality. This REST API exposes a Cypher endpoint. Using the Neo4j-Java-Rest-Binding (which wraps the Neo4j Java API around the REST calls) to interact transparently with the server, Spring Data Neo4j can work easily with the server.

By depending on `org.springframework.data:spring-data-neo4j-rest` and changing the setup to point to the remote URL of the server, we can use Spring Data Neo4j with a server installation (Example 7-26). Please note that with the current implementation, not all calls are optimally transferred over the network API, so the server interaction for individual operations will be affected by network latency and bandwidth. It is recommended to use remotely executed queries as much as possible to reduce that impact.

Example 7-26. Server connection configuration setup

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:neo4j="http://www.springframework.org/schema/data/neo4j"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/data/neo4j
        http://www.springframework.org/schema/data/neo4j/spring-neo4j.xsd">


    <neo4j:config graphDatabaseService="graphDatabaseService" />
    <bean id="graphDatabaseService"
         class="org.springframework.data.neo4j.rest.SpringRestGraphDatabase">
         <constructor-arg index="0" value="http://localhost:7474/db/data" />
    </bean>
</beans>
```

The `SpringRestGraphDatabase` connects via a `RestAPI` instance, which you can get to execute individual or batched REST operations more efficiently. For instance, creating entities with immediate property population, both for conventional or unique entities, is more efficient with the `RestAPI`.

Continuing From Here

This chapter presented some of the possibilities that graph databases—in particular Neo4j—offer and how Spring Data Neo4j gives you convenient access to them while keeping the doors open for raw, low-level graph processing.

The next thing you should do is consider the data you are working with (or want to work with) and see how connected the entities are. Look closely—you'll see they're often much more connected than you'd think at first glance. Taking one of these domains, and putting it first on a whiteboard and then into a graph database, is your first step toward realizing the power behind these concepts. For writing an application that uses the connected data, Spring Data Neo4j is an easy way to get started. It enables you

to easily create graph data and expose results of graph queries as your well-known POJOs, which eases the integration with other libraries and (UI) frameworks.

To learn how that process works for a complete web application, see [Hunger12] in the Bibliography, which is part of the reference documentation and the GitHub repository. The tutorial is a comprehensive walkthrough of creating the social movie database *cineasts.net*, and explains data modeling, integration with external services, and the web layer.

Feel free to reach out at any time to the Springsource Forums, Stackoverflow, or the Neo4j Google Group for answers to your questions. Enjoy!

CHAPTER 8

Redis: A Key/Value Store

In this chapter, we'll look at the support Spring Data offers for the key/value store Redis. We'll briefly look at how Redis manages data, show how to install and configure the server, and touch on how to interact with it from the command line. Then we'll look at how to connect to the server from Java and how the `RedisTemplate` organizes the multitude of operations we can perform on data stored in Redis. We'll look at ways to store POJOs using JSON, and we'll also briefly discuss how to use the fast and efficient pub/sub (publish/subscribe) capability to do basic event-based programming.

Redis in a Nutshell

Redis is an extremely high-performance, lightweight data store. It provides key/value data access to persistent byte arrays, lists, sets, and hash data structures. It supports atomic counters and also has an efficient topic-based pub/sub messaging functionality. Redis is simple to install and run and is, above all, very, very fast at data access. What it lacks in complex querying functionality (like that found in Riak or MongoDB), it makes up for in speed and efficiency. Redis servers can also be clustered together to provide for very flexible deployment. It's easy to interact with Redis from the command line using the *redis-cli* binary that comes with the installation.

Setting Up Redis

To start working with Redis, you'll want to have a local installation. Depending on your platform, the installation process ranges from easy to literally one command. The easiest installation process, shown in Example 8-1, is on Mac OS X using Homebrew. Other Unix systems are natively supported if you build the server from source. (Build instructions are on the Redis website, though they are identical to most other *NIX packages we've built—namely, unzip it, `cd` into that directory, and type **`make`**.) The download page for Redis also lists a couple of unofficial efforts to port Redis to the Win32/64 platform, though those are not considered production quality. For the purposes of this chapter, we'll stick to the *NIX version, where Redis is most at home.

Example 8-1. Installing Redis on Mac OS X using Homebrew

```
$ brew install redis
==> Downloading http://redis.googlecode.com/files/redis-2.4.15.tar.gz
######################################################################## 100.0%
==> make -C /private/tmp/homebrew-redis-2.4.15-WbS5/redis-2.4.15/src CC=/usr/bin/clang
==> Caveats
If this is your first install, automatically load on login with:
    mkdir -p ~/Library/LaunchAgents
    cp /usr/local/Cellar/redis/2.4.15/homebrew.mxcl.redis.plist ~/Library/LaunchAgents/
    launchctl load -w ~/Library/LaunchAgents/homebrew.mxcl.redis.plist

If this is an upgrade and you already have the homebrew.mxcl.redis.plist loaded:
    launchctl unload -w ~/Library/LaunchAgents/homebrew.mxcl.redis.plist
    cp /usr/local/Cellar/redis/2.4.15/homebrew.mxcl.redis.plist ~/Library/LaunchAgents/
    launchctl load -w ~/Library/LaunchAgents/homebrew.mxcl.redis.plist

  To start redis manually:
    redis-server /usr/local/etc/redis.conf

  To access the server:
    redis-cli
==> Summary
/usr/local/Cellar/redis/2.4.15: 9 files, 556K, built in 12 seconds
```

Just so we can get a server running quickly and see some results, let's run the server in a terminal, in the foreground. This is good for debugging because it logs directly to the console to let you know what the server is doing internally. Instructions on installing a boot script to get the server running when you restart your machine will, of course, vary by platform. Setting that up is an exercise left to the reader.

We're just going to use the default settings for the server, so starting it is simply a matter of executing `redis-server`, as in Example 8-2.

Example 8-2. Starting the server

```
$ redis-server
[91688] 25 Jul 09:37:36 # Warning: no config file specified, using the default config.
  In order to specify a config file use 'redis-server /path/to/redis.conf'
[91688] 25 Jul 09:37:36 * Server started, Redis version 2.4.15
[91688] 25 Jul 09:37:36 * The server is now ready to accept connections on port 6379
[91688] 25 Jul 09:37:36 - 0 clients connected (0 slaves), 922304 bytes in use
```

Using the Redis Shell

Redis comes with a very useful command-line shell that you can use interactively or from batch jobs. We'll just be using the interactive part of the shell so we can poke around inside the server, look at our data, and interact with it. The command shell has an extensive help system (Example 8-3) so once you're in there, hit the Tab key a couple of times to have the shell prompt you for help.

Example 8-3. Interacting with the Redis server

```
$ redis-cli
redis 127.0.0.1:6379> help
redis-cli 2.4.15
Type: "help @<group>" to get a list of commands in <group>
      "help <command>" for help on <command>
      "help <tab>" to get a list of possible help topics
      "quit" to exit
redis 127.0.0.1:6379> |
```

The Redis documentation is quite helpful here, as it gives a nice overview of all the commands available and shows you some example usage. Keep this page handy because you'll be referring back to it often.

It will pay dividends to spend some time familiarizing yourself with the basic SET and GET commands. Let's take a moment and play with inserting and retrieving data (Example 8-4).

Example 8-4. SET and GET data in Redis

```
$ redis-cli
redis 127.0.0.1:6379> keys *
(empty list or set)
redis 127.0.0.1:6379> set spring-data-book:redis:test-value 1
OK
redis 127.0.0.1:6379> keys *
1) "spring-data-book:redis:test-value"
redis 127.0.0.1:6379> get spring-data-book:redis:test-value
"1"
redis 127.0.0.1:6379> |
```

Notice that we didn't put quotes around the value `1` when we SET it. Redis doesn't have datatypes like other datastores, so it sees every value as a list of bytes. In the command shell, you'll see these printed as strings. When we GET the value back out, we see `"1"` in the command shell. We know by the quotes, then, that this is a string.

Connecting to Redis

Spring Data Redis supports connecting to Redis using either the Jedis, JRedis, RJC, or SRP driver libraries. Which you choose doesn't make any difference to your use of the Spring Data Redis library. The differences between the drivers have been abstracted out into a common set of APIs and template-style helpers. For the sake of simplicity, the example project uses the Jedis driver.

To connect to Redis using Jedis, you need to create an instance of `org.springframework.data.redis.connection.jedis.JedisConnectionFactory`. The other driver libraries have corresponding `ConnectionFactory` subclasses. A configuration using JavaConfig might look like Example 8-5.

Example 8-5. Connecting to Redis with Jedis

```
@Configuration
public class ApplicationConfig {

  @Bean
  public JedisConnectionFactory connectionFactory() {
    JedisConnectionFactory connectionFactory = new JedisConnectionFactory();
    connectionFactory.setHostName("localhost");
    connectionFactory.setPort(6379);
    return connectionFactory;
  }
}
```

The central abstraction you're likely to use when accessing Redis via Spring Data Redis is the `org.springframework.data.redis.core.RedisTemplate` class. Since the feature set of Redis is really too large to effectively encapsulate into a single class, the various operations on data are split up into separate `Operations` classes as follows (names are self-explanatory):

- `ValueOperations`
- `ListOperations`
- `SetOperations`
- `ZSetOperations`
- `HashOperations`
- `BoundValueOperations`
- `BoundListOperations`
- `BoundSetOperations`
- `BoundZSetOperations`
- `BoundHashOperations`

Object Conversion

Because Redis deals directly with byte arrays and doesn't natively perform `Object` to `byte[]` translation, the Spring Data Redis project provides some helper classes to make it easier to read and write data from Java code. By default, all keys and values are stored as serialized Java objects. If you're going to be dealing largely with `String`s, though, there is a template class—`StringRedisTemplate`, shown in Example 8-6—that installs the `String` serializer and has the added benefit of making your keys and values human-readable from the Redis command-line interface.

Example 8-6. Using the StringRedisTemplate

```
@Configuration
public class ApplicationConfig {
```

```
  @Bean
  public JedisConnectionFactory connectionFactory() { … }

  @Bean
  public StringRedisTemplate redisTemplate() {
    StringRedisTemplate redisTemplate = new StringRedisTemplate();
    redisTemplate.setConnectionFactory(connectionFactory());
    return redisTemplate;
  }
}
```

To influence how keys and values are serialized and deserialized, Spring Data Redis provides a `RedisSerializer` abstraction that is responsible for actually reading and writing the bytes stored in Redis. Set an instance of `org.springframework.data.redis.serializer.RedisSerializer` on either the `keySerializer` or `valueSerializer` property of the template. There is already a built-in `RedisSerializer` for `String`s, so to use `String`s for keys and `Long`s for values, you would create a simple serializer for `Long`s, as shown in Example 8-7.

Example 8-7. Creating reusable serializers

```
public enum LongSerializer implements RedisSerializer<Long> {

  INSTANCE;

  @Override
  public byte[] serialize(Long aLong) throws SerializationException {
    if (null != aLong) {
      return aLong.toString().getBytes();
    } else {
      return new byte[0];
    }
  }

  @Override
  public Long deserialize(byte[] bytes) throws SerializationException {
    if (bytes.length > 0) {
      return Long.parseLong(new String(bytes));
    } else {
      return null;
    }
  }
}
```

To use these serializers to make it easy to do type conversion when working with Redis, set the `keySerializer` and `valueSerializer` properties of the template like in the snippet of JavaConfig code shown in Example 8-8.

Example 8-8. Using serializers in a template instance

```
@Bean
public RedisTemplate<String, Long> longTemplate() {
```

```
  private static final StringRedisSerializer STRING_SERIALIZER =
    new StringRedisSerializer();

  RedisTemplate<String, Long> tmpl = new RedisTemplate<String, Long>();
  tmpl.setConnectionFactory(connFac);
  tmpl.setKeySerializer(STRING_SERIALIZER);
  tmpl.setValueSerializer(LongSerializer.INSTANCE);

  return tmpl;
}
```

You're now ready to start storing counts in Redis without worrying about `byte[]`-to-`Long` conversion. Since Redis supports such a large number of operations—which makes for a lot of methods on the helper classes—the methods for getting and setting values are defined in the various `RedisOperations` interfaces. You can access each of these interfaces by calling the appropriate `opsForX` method on the `RedisTemplate`. Since we're only storing discrete values in this example, we'll be using the `ValueOperations` template (Example 8-9).

Example 8-9. Automatic type conversion when setting and getting values

```
public class ProductCountTracker {

  @Autowired
  RedisTemplate<String, Long> redis;

  public void updateTotalProductCount(Product p) {
    // Use a namespaced Redis key
    String productCountKey = "product-counts:" + p.getId();

    // Get the helper for getting and setting values
    ValueOperations<String, Long> values = redis.opsForValue();

    // Initialize the count if not present
    values.setIfAbsent(productCountKey, 0L);

    // Increment the value by 1
    Long totalOfProductInAllCarts = values.increment(productCountKey, 1);
  }

}
```

After you call this method from your application and pass a `Product` with an `id` of `1`, you should be able to inspect the value from `redis-cli` and see the string `"1"` by issuing the Redis command `get product-counts:1`.

Object Mapping

It's great to be able to store simple values like counters and strings in Redis, but it's often necessary to store richer sets of related information. In some cases, these might be properties of an object. In other cases, they might be the keys and values of a hash.

Using the `RedisSerializer`, you can store an object into Redis as a single value. But doing so won't make the properties of that object very easy to inspect or retrieve individually. What you probably want in that case is to use a Redis hash. Storing your properties in a hash lets you access all of those properties together by pulling them all out as a `Map<String, String>`, or you can reference the individual properties in the hash without touching the others.

Since everything in Redis is a `byte[]`, for this hash example we're going to simpify by using `String`s for keys and values. The operations for hashes, like those for values, sets, and so on, are accessible from the `RedisTemplate opsForHash()` method. See Example 8-10.

Example 8-10. Using the HashOperations interface

```
private static final RedisSerializer<String> STRING_SERIALIZER =
  new StringRedisSerializer();

public void updateTotalProductCount(Product p) {

  RedisTemplate tmpl = new RedisTemplate();
  tmpl.setConnectionFactory(connectionFactory);
  // Use the standard String serializer for all keys and values
  tmpl.setKeySerializer(STRING_SERIALIZER);
  tmpl.setHashKeySerializer(STRING_SERIALIZER);
  tmpl.setHashValueSerializer(STRING_SERIALIZER);

  HashOperations<String, String, String> hashOps = tmpl.opsForHash();

  // Access the attributes for the Product
  String productAttrsKey = "products:attrs:" + p.getId();

  Map<String, String> attrs = new HashMap<String, String>();

  // Fill attributes
  attrs.put("name", "iPad");
  attrs.put("deviceType", "tablet");
  attrs.put("color", "black");
  attrs.put("price", "499.00");

  hashOps.putAll(productAttrsKey, attrs);

}
```

Assuming the `Product` has an `id` of `1`, from `redis-cli` you should be able to list all the keys of the hash by using the `HKEYS` command (Example 8-11).

Example 8-11. Listing hash keys

```
redis 127.0.0.1:6379> hkeys products:attrs:1
1) "price"
2) "color"
3) "deviceType"
4) "name"
```

```
redis 127.0.0.1:6379> hget products:attrs:1 name
"iPad"
```

Though this example just uses a `String` for the hash's value, you can use any `RedisSerializer` instance for the template's `hashValueSerializer`. If you wanted to store complex objects rather than `String`s, for instance, you might replace the `hashValueSerializer` in the template with an instance of `org.springframework.data.redis.serializer.JacksonJsonRedisSerializer` for serializing objects to JSON, or `org.springframework.data.redis.serializer.OxmSerializer` for marshalling and unmarshalling your object using Spring OXM.

Atomic Counters

Many people choose to use Redis because of the atomic counters that it supports. If multiple applications are all pointing at the same Redis instance, then those distributed applications can consistently and atomically increment a counter to ensure uniqueness. Java already contains `AtomicInteger` and `AtomicLong` classes for atomically incrementing counters across threads, but that won't help us if those counters are in other JVM processes or `ClassLoader`s. Spring Data Redis implements a couple of helper classes similar to `AtomicInteger` and `AtomicLong` and backs them by a Redis instance. Accessing distributed counters within your application is as easy as creating an instance of these helper classes and pointing them all to the same Redis server (Example 8-12).

Example 8-12. Using RedisAtomicLong

```
public class CountTracker {

  @Autowired
  RedisConnectionFactory connectionFactory;

  public void updateProductCount(Product p) {
    // Use a namespaced Redis key
    String productCountKey = "product-counts:" + p.getId();

    // Create a distributed counter.
    // Initialize it to zero if it doesn't yet exist
    RedisAtomicLong productCount =
      new RedisAtomicLong(productCountKey, connectionFactory, 0);

    // Increment the count
    Long newVal = productCount.incrementAndGet();
  }

}
```

Pub/Sub Functionality

Another important benefit of using Redis is the simple and fast publish/subscribe functionality. Although it doesn't have the advanced features of a full-blown message broker, Redis' pub/sub capability can be used to create a lightweight and flexible event bus. Spring Data Redis exposes a couple of helper classes that make working with this functionality extremely easy.

Listening and Responding to Messages

Following the pattern of the JMS `MessageListenerAdapter`, Spring Data Redis has a `MessageListenerAdapter` abstraction that works in basically the same way (Example 8-13). The JMS version, the `MessageListenerAdapter`, is flexible in what kind of listeners it accepts if you don't want to be tied to a particular interface. You can pass a POJO with a `handleMessage` method that takes as its first argument an `org.springframework.data.redis.connection.Message`, a `String`, a `byte[]`, or, if you use an appropriate `RedisSerializer`, an object of any convertible type. You can define an optional second parameter, which will be the channel or pattern that triggered this invocation. There is also a `MessageListener` interface to give your beans a solid contract to implement if you want to avoid the reflection-based invocation that's done when passing in a POJO.

Example 8-13. Adding a simple MessageListener using JavaConfig

```
@Bean public MessageListener dumpToConsoleListener() {
  return new MessageListener() {
    @Override
    public void onMessage(Message message, byte[] pattern) {
      System.out.println("FROM MESSAGE: " + new String(message.getBody()));
    }
  };
}
```

Spring Data Redis allows you to place POJOs on the `MessageListenerAdapter`, and the container will convert the incoming message into your custom type using a converter you provide. (See Example 8-14.)

Example 8-14. Setting up a MessageListenerContainer and simple message listener using a POJO

```
@Bean MessageListenerAdapter beanMessageListener() {
  MessageListenerAdapter listener = new MessageListenerAdapter( new BeanMessageListener()
);
  listener.setSerializer( new BeanMessageSerializer() );
  return listener;
}
```

`BeanMessageListener`, shown in Example 8-15, is simply a POJO with a method named `handleMessage` defined on it, with the first parameter being of type `BeanMessage` (an arbitrary class we've created for this example). It has a single property on it called `message`. Our `RedisSerializer` will store the contents of this `String` as bytes.

Example 8-15. Adding a POJO listener using JavaConfig

```
public class BeanMessageListener {
  public void handleMessage( BeanMessage msg ) {
    System.out.println( "msg: " + msg.message );
  }
}
```

The component responsible for actually invoking your listeners when the event is triggered is an `org.springframework.data.redis.listener.RedisMessageListenerContainer`. As demonstrated in Example 8-16, it needs to be configured with a `RedisConnectionFactory` and a set of listeners. The container has life cycle methods on it that will be called by the Spring container if you create it inside an `ApplicationContext`. If you create this container programmatically, you'll need to call the `afterPropertiesSet()` and `start()` methods manually. Remember to assign your listeners before you call the `start()` method, though, or your handlers will not be invoked since the wiring is done in the `start()` method.

Example 8-16. Configuring a RedisMessageListenerContainer

```
@Bean RedisMessageListenerContainer container() {
  RedisMessageListenerContainer container = new RedisMessageListenerContainer();
  container.setConnectionFactory(redisConnectionFactory());
  // Assign our BeanMessageListener to a specific channel
  container.addMessageListener(beanMessageListener(),
    new ChannelTopic("spring-data-book:pubsub-test:dump"));
  return container;
}
```

Using Spring's Cache Abstraction with Redis

Spring 3.1 introduced a common and reusable caching abstraction. This makes it easy to cache the results of method calls in your POJOs without having to explicitly manage the process of checking for the existence of a cache entry, loading new ones, and expiring old cache entries. Spring 3.1 gives you some helpers that work with a variety of cache backends to perform these functions for you.

Spring Data Redis supports this generic caching abstraction with the `org.springframework.data.redis.cache.RedisCacheManager`. To designate Redis as the backend for using the caching annotations in Spring, you just need to define a `RedisCacheManager` bean in your `ApplicationContext`. Then annotate your POJOs like you normally would, with `@Cacheable` on methods you want cached.

The `RedisCacheManager` needs a configured `RedisTemplate` in its constructor. In this example, we're letting the caching abstraction generate a unique integer for us to serve as the cache key. There are lots of options for how the cache manager stores your results. You can configure this behavior by placing the appropriate annotation on your `@Cacheable` methods. In Example 8-17, we're using an integer serializer for the key and the built-in `JdkSerializationRedisSerializer` for the value, since we really don't know

what we'll be storing. Using JDK serialization allows us to cache any `Serializable` Java object.

To enable the caching interceptor in your `ApplicationContext` using JavaConfig, you simply put the `@EnableCaching` annotation on your `@Configuration`.

Example 8-17. Configuring caching with RedisCacheManager

```
@Configuration
@EnableCaching
public class CachingConfig extends ApplicationConfig {

  @SuppressWarnings({"unchecked"})
  @Bean public RedisCacheManager redisCacheManager() {
    RedisTemplate tmpl = new RedisTemplate();
    tmpl.setConnectionFactory( redisConnectionFactory() );
    tmpl.setKeySerializer( IntSerializer.INSTANCE );
    tmpl.setValueSerializer( new JdkSerializationRedisSerializer() );
    RedisCacheManager cacheMgr = new RedisCacheManager( tmpl );
    return cacheMgr;
  }

  @Bean public CacheableTest cacheableTest() {
    return new CacheableTest();
  }

}
```

PART IV

Rapid Application Development

CHAPTER 9

Persistence Layers with Spring Roo

Spring Roo is a rapid application development tool for Java developers. With Roo, you can easily build full Java applications in minutes.

We won't be covering all aspects of Roo development in this chapter. We will focus on the new repository support for JPA and MongoDB that uses Spring Data to provide this support. If you want to read more about Roo, go to the Spring Roo project home page, where you can find links to the reference manual. While on the project home page, look for a link to download a free O'Reilly ebook by Josh Long and Steve Mayzak called *Getting Started with Roo* [LongMay11]. This ebook covers an older 1.1 version of Roo that does not support the repository layer, but it is a good introduction to Roo in general. The most up-to-date guide for using Spring Roo is *Spring Roo in Action* by Ken Rimple and Srini Penchikala [RimPen12].

A Brief Introduction to Roo

Roo works its magic using code generation combined with AspectJ for injecting behavior into your domain and web classes. When you work on a Roo project, the project files are monitored by Roo and additional artifacts are generated. You still have your regular Java classes that you can edit, but there are additional features provided for free. When you create a class with Roo and annotate that class with one or more annotations that provide additional capabilities, Roo will generate a corresponding AspectJ file that contains one or more AspectJ inter type declarations (ITD). There is, for instance, an `@RooJavaBean` annotation that triggers the generation of an AspectJ aspect declaration that provides ITDs that introduce getters and setters in your Java class. There's no need to code that yourself. Let's see a quick example of how that would look. Our simple bean class in shown in Example 9-1.

Example 9-1. A simple Java bean class: Address.java

```
@RooJavaBean
public class Address {
```

```
    private String street;

    private String city;

    private String country;
}
```

As you can see, we don't code the getters and setters. They will be introduced by the backing AspectJ aspect file since we used the `@RooJavaBean` annotation. The generated AspectJ file looks like Example 9-2.

Example 9-2. The generated AspectJ aspect definition: Address_Roo_JavaBean.aj

```
// WARNING: DO NOT EDIT THIS FILE. THIS FILE IS MANAGED BY SPRING ROO.
// You may push code into the target .java compilation unit if you wish to edit any member(s).

package com.oreilly.springdata.roo.domain;

import com.oreilly.springdata.roo.domain.Address;

privileged aspect Address_Roo_JavaBean {

    public String Address.getStreet() {
        return this.street;
    }

    public void Address.setStreet(String street) {
        this.street = street;
    }

    public String Address.getCity() {
        return this.city;
    }

    public void Address.setCity(String city) {
        this.city = city;
    }

    public String Address.getCountry() {
        return this.country;
    }

    public void Address.setCountry(String country) {
        this.country = country;
    }

}
```

You can see that this is defined as a privileged aspect, which means that it will have access to any private variables declared in the target class. The way you would define ITDs is by preceding any method names with the target class name, separated by a dot. So `public String Address.getStreet()` will introduce a new method in the `Address` class with a `public String getStreet()` signature.

As you can see, Roo follows a specific naming pattern that makes it easier to identify what files it has generated. To work with Roo, you can either use a command-line shell or edit your source files directly. Roo will synchronize all changes and maintain the source and generated files as necessary.

When you ask Roo to create a project for you, it generates a *pom.xml* file that is ready for you to use when you build the project with Maven. In this *pom.xml* file, there is a Maven compile as well as an AspectJ plug-in defined. This means that all the AspectJ aspects are woven at compile time. In fact, nothing from Roo remains in the Java class files that your build generates. There is no runtime jar dependency. Also, the Roo annotations are source-level retention only, so they will not be part of your class files. You can, in fact, easily get rid of Roo if you so choose. You have the option of pushing all of the code defined in the AspectJ files into the appropriate source files and removing any of these AspectJ files. This is called *push-in refactoring*, and it will leave you with a pure Java solution, just as if you had written everything from scratch yourself. Your application still retains all of the functionality.

Roo's Persistence Layers

Spring Roo started out supporting JPA as the only persistence option. It also was opinionated in terms of the data access layer. Roo prescribed an active record data access style where each entity provides its finder, save, and delete methods.

Starting with Roo version 1.2, we have additional options for the persistence layer (see Figure 9-1). Roo now allows you to choose between the default active record style and a repository-based persistence layer. If you choose the repository approach, you have a choice between JPA and MongoDB as the persistence providers. The actual repository support that Roo uses is the one provided by Spring Data, which we have already seen in Chapter 2.

In addition to an optional repository layer, Roo now also lets you define a service layer on top of either the active record style or repository style persistence layer.

Quick Start

You can use Roo either as a command-line tool or within an IDE, like the free Spring Tool Suite, that has built-in Roo support. Another IDE that has support for Roo is IntelliJ IDEA, but we won't be covering the support here.

Using Roo from the Command Line

First, you need to download the latest Spring Roo distribution from the download page. Once you have the file downloaded, unzip it somewhere on your system. In the *bin* directory, there is a *roo.sh* shell script for Unix-style systems as well as a *roo.bat*

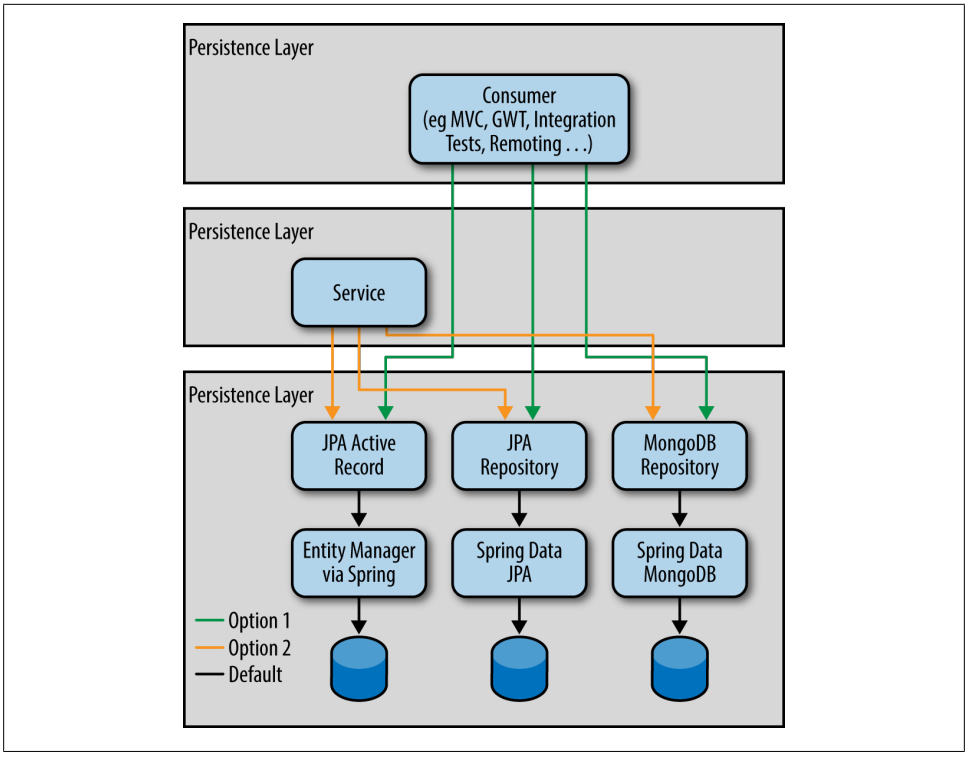


Figure 9-1. Spring Roo 1.2 layer architecture

batch file for Windows. When you want to create a Roo project, simply create a project directory and start Roo using the shell script or the batch file. If you add the *bin* directory to your path, you can just use the command name to start Roo; otherwise, you will have to provide the fully qualified path.

Once Roo starts up, you are greeted with the following screen (we entered `hint` at the prompt to get the additional information):

```
    ___ ___ ___
   / __ \/ __ \/ __ \
  / /_/ / / / / / / /
 / _, _/ /_/ / /_/ /
/_/ |_|\____/\____/    1.2.2.RELEASE [rev 7d75659]


Welcome to Spring Roo. For assistance press TAB or type "hint" then hit ENTER.
roo> hint
Welcome to Roo! We hope you enjoy your stay!

Before you can use many features of Roo, you need to start a new project.

To do this, type 'project' (without the quotes) and then hit TAB.
```

```
Enter a --topLevelPackage like 'com.mycompany.projectname' (no quotes).
When you've finished completing your --topLevelPackage, press ENTER.
Your new project will then be created in the current working directory.

Note that Roo frequently allows the use of TAB, so press TAB regularly.
Once your project is created, type 'hint' and ENTER for the next suggestion.
You're also welcome to visit http://forum.springframework.org for Roo help.
roo>
```

We are now ready to create a project and start developing our application. At any time, you can enter **`hint`**, and Roo will respond with some instruction on what to do next based on the current state of your application development. To cut down on typing, Roo will attempt to complete the commands you enter whenever you hit the Tab key.

Using Roo with Spring Tool Suite

The Spring Tool Suite comes with built-in Roo support, and it also comes bundled with Maven and the Developer Edition of *VMware vFabric tc Server*. This means that you have everything you need to develop applications with Roo. Just create your first Roo application using the menu option File→New→Spring Roo Project. You can see this in action in Figure 9-2.

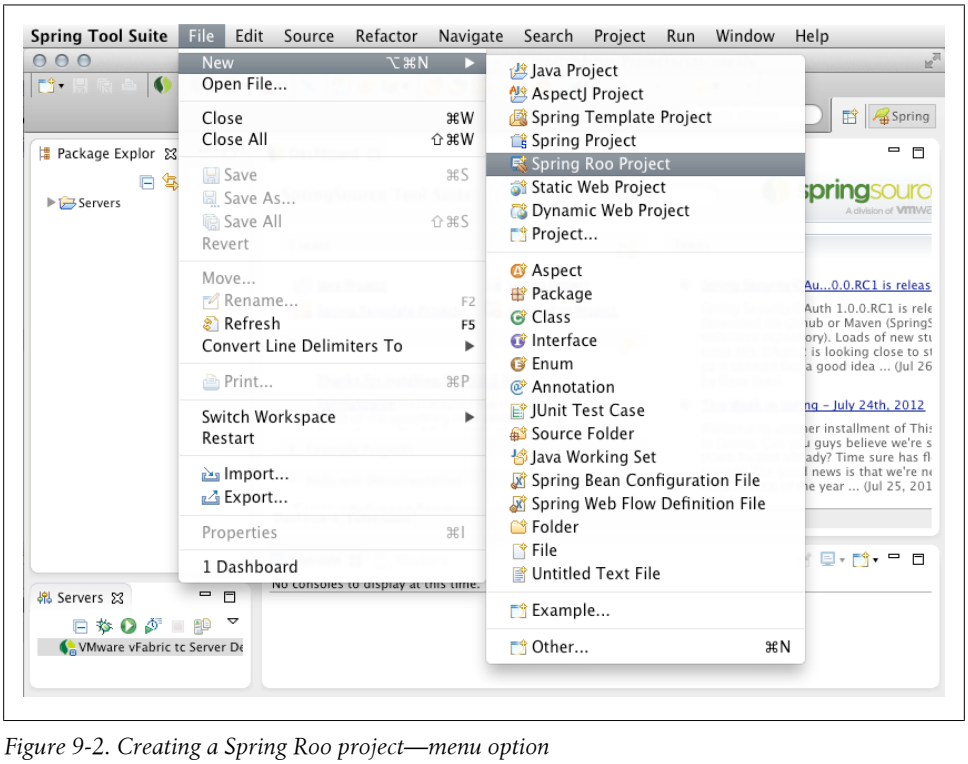


Figure 9-2. Creating a Spring Roo project—menu option

This opens a “Create a new Roo Project” dialog screen, as shown in Figure 9-3.

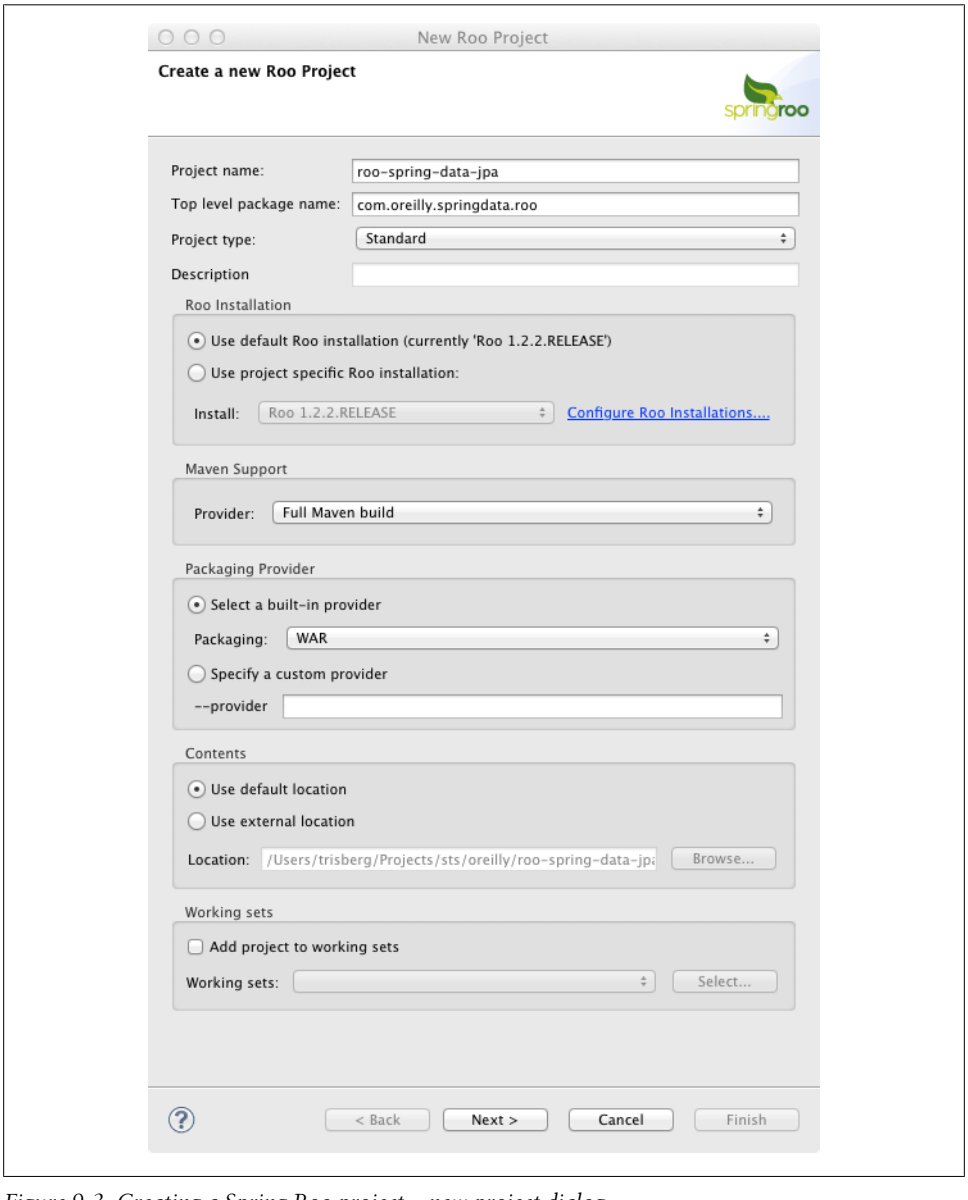


Figure 9-3. Creating a Spring Roo project—new project dialog

Just fill in the “Project name” and “Top level package name,” and then select WAR as the packaging. Click Next, and then click Finish on the next screen. The project should now be created, and you should also see the Roo shell window, as shown in Figure 9-4.

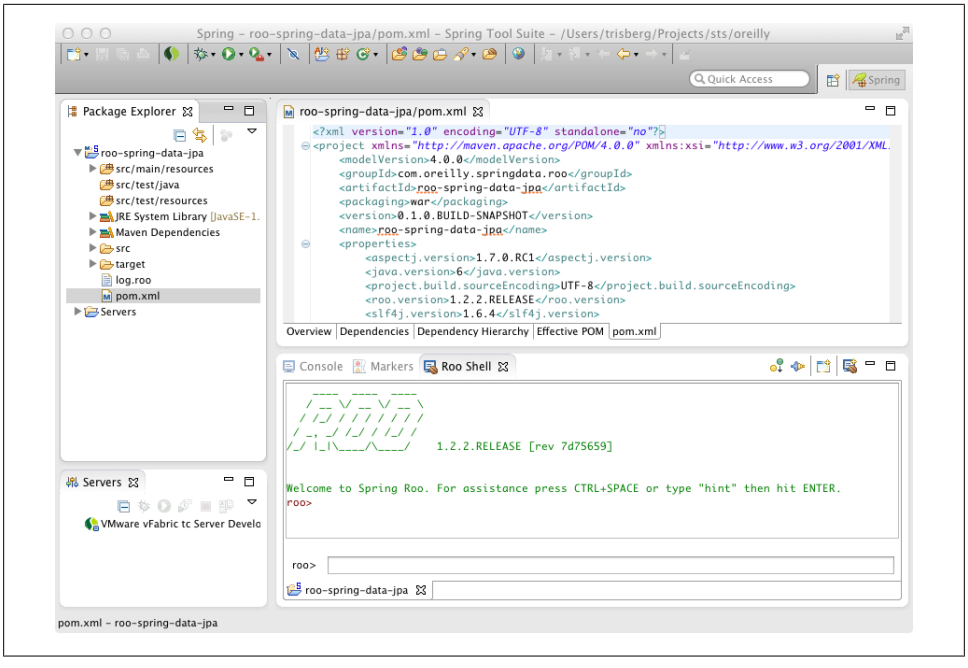


Figure 9-4. Creating a Spring Roo project—new project with Roo Shell

A Spring Roo JPA Repository Example

We are now ready to build the first Roo project. We will start with a customer service application based on the same domain model that we have seen in earlier chapters. We will create a `Customer` class and an associated `Address` class, link them together, and create repositories and really basic data entry screens for them. Since Roo's repository support supports both JPA and MongoDB, using the Spring Data repository support, we will create one of each kind of application. As you will see, they are almost identical, but there are a couple of differences that we will highlight. So, let's get started. We'll begin with the JPA application.

Creating the Project

If you are using Spring Tool Suite, then just follow the aforementioned instructions to create a new Spring Roo project. On the "Create a new Roo Project" dialog screen, provide the following settings:

- Project name: `roo-spring-data-jpa`
- Top level package name: `com.oreilly.springdata.roo`
- Packaging: `WAR`

If you are using the command-line Roo shell, you need to create a *roo-spring-data-jpa* directory; once you change to this new directory, you can start the Roo shell as just explained. At the `roo>` prompt, enter the following command:

```
project --topLevelPackage com.oreilly.springdata.roo ↪
  --projectName roo-spring-data-jpa --java 6 --packaging WAR
```

You now have created a new project, and we are ready to start developing the application. From here on, the actions will be the same whether you are using the Roo shell from the command line or inside the Spring Tool Suite.

Setting Up JPA Persistence

Setting up the JPA persistence configuration consists of selecting a JPA provider and a database. We will use Hibernate together with HSQLDB for this example. At the `roo>` prompt, enter the following:

```
jpa setup --provider HIBERNATE --database HYPERSONIC_PERSISTENT
```

Remember that when entering these commands, you can always press the Tab key to get completion and suggestions for available options. If you are using the Spring Tool Suite, press Ctrl+Space instead.

Creating the Entities

Let's create our entities, starting with the `Address` class:

```
entity jpa --class ~.domain.Address --activeRecord false
field string --fieldName street --notNull
field string --fieldName city --notNull
field string --fieldName country --notNull
```

That wasn't too hard. Note that we specified `--activeRecord false`, which means that we will have to provide the CRUD functionality using a repository. The resulting `Address` class looks like this:

```
package com.oreilly.springdata.roo.domain;

import javax.validation.constraints.NotNull;
import org.springframework.roo.addon.javabean.RooJavaBean;
import org.springframework.roo.addon.jpa.entity.RooJpaEntity;
import org.springframework.roo.addon.tostring.RooToString;

@RooJavaBean
@RooToString
@RooJpaEntity
public class Address {

    @NotNull
    private String street;
```

```
    @NotNull
    private String city;

    @NotNull
    private String country;
}
```

We see the private fields we declared and three Roo annotations: `@RooJavaBean`, `@RooToString`, and `@RooJpaEntity`. These annotations have corresponding AspectJ aspect declarations that you can find in the same directory as the Java class.

Let's create the `EmailAddress` and `Customer` classes next. The `EmailAddress` is an embeddable class with a single `value` field. We need to ask Roo to ignore the fact that *value* is a reserved word for some SQL databases. We also provide a column name of `email` since that will be more descriptive for anyone inspecting the database table. Using this `embeddable` in a field declarations, we specify it as an `embedded` field.

```
embeddable --class ~.domain.EmailAddress
field string --fieldName value --notNull --column email --permitReservedWords
entity jpa --class ~.domain.Customer --activeRecord false
field string --fieldName firstName --notNull
field string --fieldName lastName --notNull
field embedded --fieldName emailAddress --type ~.domain.EmailAddress
field set --fieldName addresses --type ~.domain.Address
```

The last command creates a many-to-many relationship to the address table, allowing us to provide a number of addresses for each customer. This is the resulting `Customer` class:

```
package com.oreilly.springdata.roo.domain;

import java.util.HashSet;
import java.util.Set;
import javax.persistence.CascadeType;
import javax.persistence.Embedded;
import javax.persistence.ManyToMany;
import javax.validation.constraints.NotNull;
import org.springframework.roo.addon.javabean.RooJavaBean;
import org.springframework.roo.addon.jpa.entity.RooJpaEntity;
import org.springframework.roo.addon.tostring.RooToString;

@RooJavaBean
@RooToString
@RooJpaEntity
public class Customer {

    @NotNull
    private String firstName;

    @NotNull
    private String lastName;

    @Embedded
    private EmailAddress emailAddress;
```

```
    @ManyToMany(cascade = CascadeType.ALL)
    private Set<Address> addresses = new HashSet<Address>();
}
```

For completeness, we'll show the `EmailAddress` class as well:

```
package com.oreilly.springdata.roo.domain;

import javax.persistence.Column;
import javax.persistence.Embeddable;
import javax.validation.constraints.NotNull;
import org.springframework.roo.addon.javabean.RooJavaBean;
import org.springframework.roo.addon.tostring.RooToString;

@RooJavaBean
@RooToString
@Embeddable
public class EmailAddress {

    @NotNull
    @Column(name = "email")
    private String value;
}
```

The most interesting part of this class is that it is `@Embeddable` and that we have defined the value property to be stored in a database column named `email`.

Defining the Repositories

With the entities in place, we can now create the JPA repositories. We create one repository for each entity.

```
repository jpa --interface ~.repository.CustomerRepository --entity ~.domain.Customer
repository jpa --interface ~.repository.AddressRepository --entity ~.domain.Address
```

At this point we could also create a service layer, but since this is such a simple application, we'll skip this step.

Creating the Web Layer

Now we need some really simple web pages so we can enter and modify our customer and address data. We'll just stick with the screens generated by Roo.

```
web mvc setup
web mvc scaffold --class ~.web.CustomerController --backingType ~.domain.Customer
web mvc scaffold --class ~.web.AddressController --backingType ~.domain.Address
```

There is one thing we have to do. Roo doesn't know how to map the `EmailAddress` class between the `String` representation used for web pages and the `EmailAddress` type used for persistence. We need to add converters to the `ApplicationConversionServiceFactoryBean` that Roo generated; Example 9-3 shows how.

Example 9-3. The generated ApplicationConversionServiceFactoryBean.java with converters added

```
package com.oreilly.springdata.roo.web;

import org.springframework.core.convert.converter.Converter;
import org.springframework.format.FormatterRegistry;
import org.springframework.format.support.FormattingConversionServiceFactoryBean;
import org.springframework.roo.addon.web.mvc.controller.converter.RooConversionService;

import com.oreilly.springdata.roo.domain.EmailAddress;

/**
 * A central place to register application converters and formatters.
 */
@RooConversionService
public class ApplicationConversionServiceFactoryBean
        extends FormattingConversionServiceFactoryBean {

  @Override
  protected void installFormatters(FormatterRegistry registry) {
    super.installFormatters(registry);
    // Register application converters and formatters
    registry.addConverter(getStringToEmailAddressConverter());
    registry.addConverter(getEmailAddressConverterToString());
  }

  public Converter<String, EmailAddress> getStringToEmailAddressConverter() {
    return new Converter<String, EmailAddress>() {
      @Override
      public EmailAddress convert(String source) {
        EmailAddress emailAddress = new EmailAddress();
        emailAddress.setAddress(source);
        return emailAddress;
      }
    };
  }

  public Converter<EmailAddress, String> getEmailAddressConverterToString() {
    return new Converter<EmailAddress, String>() {
      @Override
      public String convert(EmailAddress source) {
        return source.getAddress();
      }
    };
  }
}
```

Running the Example

Now we are ready to build and deploy this example. For Spring Tool Suite, just drag the application to the tc server instance and start the server. If you use the command line, simply exit the Roo shell and from the command line run the following Maven commands:

```
mvn clean package
mvn tomcat:run
```

You should now be able to open a browser and navigate to *http://localhost:8080/roo-spring-data-jpa/* and see the screen shown in Figure 9-5.

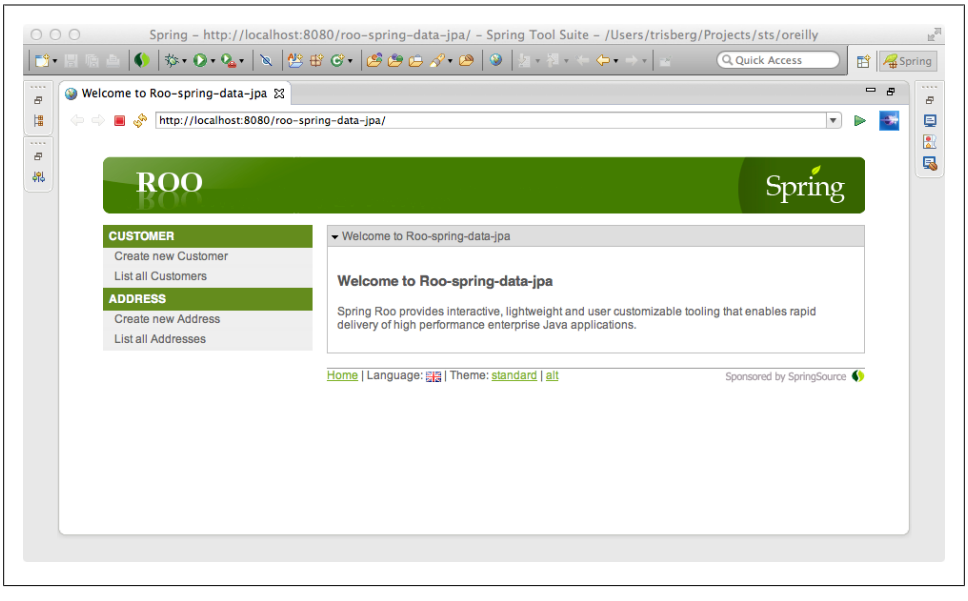


Figure 9-5. The JPA application

Our application is now complete, and we can add some addresses and then a customer or two.

If you get tired of losing your data every time you restart your app server, you can change the schema creation properties in *src/main/resources/META-INF/persistence.xml*. Change `<property name="hibernate.hbm2ddl.auto" value="create" />` to have a value of `"update"`.

A Spring Roo MongoDB Repository Example

Since Spring Data includes support for MongoDB repositories, we can use MongoDB as a persistence option when using Roo. We just won't have the option of using the active record style for the persistence layer; we can only use the repositories. Other than this difference, the process is very much the same as for a JPA solution.

Creating the Project

If you are using Spring Tool Suite, then just follow the aforementioned instructions to create a new Spring Roo project. On the "Create a new Roo Project" dialog screen, provide the following settings:

- Project name: `roo-spring-data-mongo`
- Top level package name: `com.oreilly.springdata.roo`
- Packaging: `WAR`

When using the command-line Roo shell, create a *roo-spring-data-mongo* directory. Change to this new directory and then start the Roo Shell as previously explained. At the `roo>` prompt, enter the following command:

```
project --topLevelPackage com.oreilly.springdata.roo ↪
  --projectName roo-spring-data-mongo --java 6 --packaging WAR
```

Setting Up MongoDB Persistence

Setting up the persistence configuration for MongoDB is simple. We can just accept the defaults. If you wish, you can provide a host, port, username, and password, but for a default local MongoDB installation the defaults work well. So just enter the following:

```
mongo setup
```

Creating the Entities

When creating the entities, we don't have the option of using the active record style, so there is no need to provide an `--activeRecord` parameter to opt out of it. Repositories are the default, and the only option for the persistence layer with MongoDB. Again, we start with the `Address` class:

```
entity mongo --class ~.domain.Address
field string --fieldName street --notNull
field string --fieldName city --notNull
field string --fieldName country --notNull
```

That looks very similar to the JPA example. When we move on to the `Customer` class, the first thing you'll notice that is different is that with MongoDB you don't use an `embeddable` class. That is available only for JPA. With MongoDB, you just create a plain `class` and specify `--rooAnnotations true` to enable the `@RooJavaBean` support. To use this class, you specify the field as `other`. Other than these minor differences, the entity declaration is very similar to the JPA example:

```
class --class ~.domain.EmailAddress --rooAnnotations true
field string --fieldName value --notNull --permitReservedWords
entity mongo --class ~.domain.Customer
field string --fieldName firstName --notNull
field string --fieldName lastName --notNull
```

```
field other --fieldName emailAddress --type ~.domain.EmailAddress
field set --fieldName addresses --type ~.domain.Address
```

Defining the Repositories

We declare the MongoDB repositories the same way as the JPA repositories except for the `mongo` keyword:

```
repository mongo --interface ~.repository.CustomerRepository ↪
  --entity ~.domain.Customer
repository mongo --interface ~.repository.AddressRepository --entity ~.domain.Address
```

Creating the Web Layer

The web layer is exactly the same as for the JPA example:

```
web mvc setup
web mvc scaffold --class ~.web.CustomerController --backingType ~.domain.Customer
web mvc scaffold --class ~.web.AddressController --backingType ~.domain.Address
```

Don't forget to add the converters to the `ApplicationConversionServiceFactoryBean` like we did for JPA in Example 9-3.

Running the Example

Now we are ready to build and deploy this example. This is again exactly the same as the JPA example, except that we need to have MongoDB running on our system. See Chapter 6 for instructions on how to install and run MongoDB.

For Spring Tool Suite, just drag the application to the tc server instance and start the server. If you use the command line, simply exit the Roo shell and from the command line run the following Maven commands:

```
mvn clean package
mvn tomcat:run
```

You should now be able to open a browser and navigate to *http://localhost:8080/roo-spring-data-mongo/* and see the screen in Figure 9-6.

Our second example application is now complete, and we can add some addresses and then a customer or two.

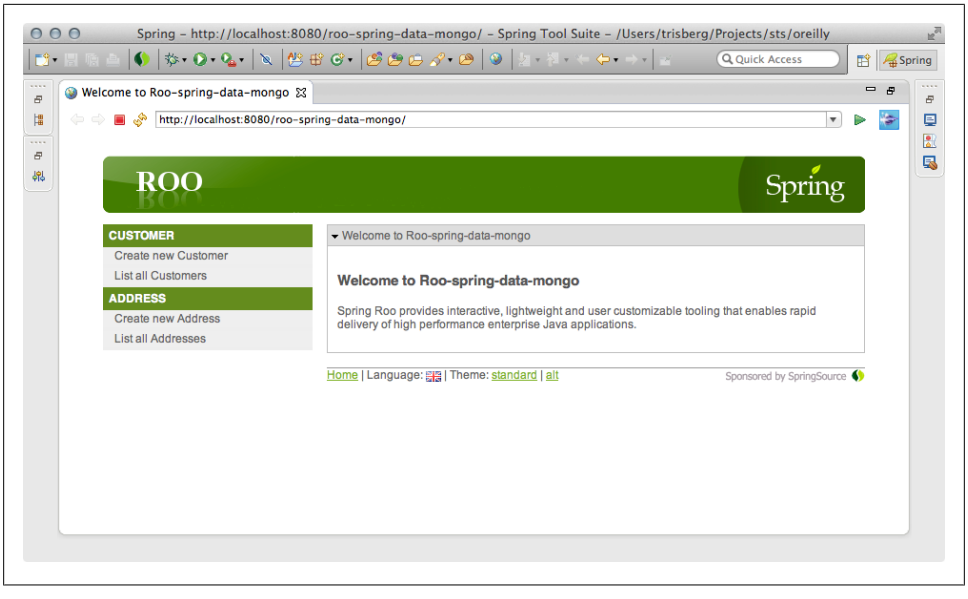


Figure 9-6. The MongoDB application

CHAPTER 10
REST Repository Exporter

When you are working with the Spring Data repository abstraction (see Chapter 2 for details), the repository interface managing an entity becomes the central point of access for it. Using the Spring Data REST repository exporter project, you can now export (as the name suggests) these managed entities via a REST web service to easily interact with the data. The exporter mechanism will transparently expose a resource per repository, map CRUD operations onto HTTP methods for that resource, and provide a means to execute query methods exposed on the repository interface.

What Is REST?

Representational State Transfer (REST) is an architectural style initially described by Roy Fielding in his dissertation analyzing styles of network-based software architectures [Fielding00]. It is a generalization of the principles behind the HTTP protocol, from which it derives the following core concepts:

Resources
: Systems expose resources to other systems: an order, a customer, and so on.

Identifiers
: These resources can be addressed through an identifier. In the HTTP world, these identifiers are URIs.

Verbs
: Each resource can be accessed and manipulated though a well-defined set of verbs. These verbs have dedicated semantics and have to be used according to those. In HTTP the commonly used verbs are `GET`, `PUT`, `POST`, `DELETE`, `HEAD`, and `OPTIONS`, as well as the rather seldomly used (or even unused) ones, `TRACE` and `CONNECT`. Not every resource has to support all of the verbs just listed, but it is required that you don't set up special verbs per resource.

Representations
: A client never interacts with the resource directly but rather through representations of it. Representations are defined through media types, which clearly identify the structure of the representation. Commonly used media types include rather

general ones like `application/xml` and `application/json`, and more structured ones like `application/atom+xml`.

Hypermedia
: The representations of a resource usually contain links to point to other resources, which allows the client to navigate the system based on the resource state and the links provided. This concept is described as Hypermedia as the Engine of Application State (HATEOAS).

Web services built based on these concepts have proven to be scalable, reliable, and evolvable. That's why REST web services are a ubiquitous means to integrate software systems. While Fielding's dissertation is a nice read, we recommend also looking at *REST in Practice*, by Jim Webber, Savas Parastatidis, and Ian Robinson. It provides a very broad, detailed, and real-world-example-driven introduction to the topic [WePaRo10].

We will have a guided tour through that functionality by walking through the *rest* module of the sample project. It is a Servlet 3.0–compliant, Spring-based web application. The most important dependency of the project is the *spring-data-rest-webmvc* library, which provides Spring MVC integration to export Spring Data JPA repositories to the Web. Currently, it works for JPA-backed repositories only, but support for other stores is on the roadmap already. The basic Spring infrastructure of the project is very similar to the one in Chapter 4, so if you haven't had a glance at that chapter already, please do so now to understand the basics.

The Sample Project

The easiest way to run the sample app is from the command line using the Maven Jetty plug-in. Jetty is a tiny servlet container capable of running Servlet 3.0 web applications. The Maven plug-in will allow you to bootstrap the app from the module folder using the command shown in Example 10-1.

Example 10-1. Running the sample application from the command line

```
$ mvn jetty:run -Dspring.profiles.active=with-data

[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Spring Data Book - REST exporter 1.0.0.BUILD-SNAPSHOT
[INFO] ------------------------------------------------------------------------
…
[INFO] <<< jetty-maven-plugin:8.1.5.v20120716:run (default-cli) @ spring-data-book-rest <<<
[INFO]
[INFO] --- jetty-maven-plugin:8.1.5.v20120716:run (default-cli) @ spring-data-book-rest ---
[INFO] Configuring Jetty for project: Spring Data Book - REST exporter
[INFO] webAppSourceDirectory …/spring-data-book/rest/src/main/webapp does not exist. \
       Defaulting to …/spring-data-book/rest/src/main/webapp
[INFO] Reload Mechanic: automatic
```

```
[INFO] Classes = …/spring-data-book/rest/target/classes
[INFO] Context path = /
[INFO] Tmp directory = …/spring-data-book/rest/target/tmp
[INFO] Web defaults = org/eclipse/jetty/webapp/webdefault.xml
[INFO] Web overrides =  none
[INFO] web.xml file = null
[INFO] Webapp directory = …/spring-data-book/rest/src/main/webapp
2012-07-31 17:58:01.709:INFO:oejs.Server:jetty-8.1.5.v20120716
2012-07-31 17:58:03.769:INFO:oejpw.PlusConfiguration:No Transaction manager found \
       - if your webapp requires one, please configure one.
2012-07-31 17:58:10.641:INFO:/:Spring WebApplicationInitializers detected on classpath: \
       [com.oreilly.springdata.rest.RestWebApplicationInitializer@34cbbc24]
…
2012-07-31 17:58:16,032  INFO est.webmvc.RepositoryRestExporterServlet: 444 - \
       FrameworkServlet 'dispatcher': initialization started
…
2012-07-31 17:58:17,159  INFO est.webmvc.RepositoryRestExporterServlet: 463 - \
       FrameworkServlet 'dispatcher': initialization completed in 1121 ms
2012-07-31 17:58:17.179:INFO:oejs.AbstractConnector:Started SelectChannelConnector@
    0.0.0.0:8080
[INFO] Started Jetty Server
```

The first thing to notice here is that we pipe a JVM argument, `spring.profiles.active`, into the execution. This populates the in-memory database with some sample products, customers, and orders so that we actually have some data to interact with. Maven now dumps some general activity information to the console. The next interesting line is the one at 17:58:10, which tells us that Jetty has discovered a `WebApplicationInitializer`—our `RestWebApplicationInitializer` in particular. A `WebApplicationInitializer` is the API equivalent to a *web.xml* file, introduced in the Servlet API 3.0. It allows us to get rid of the XML-based way to configure web application infrastructure components and instead use an API. Our implementation looks like Example 10-2.

Example 10-2. The RestWebApplicationInitializer

```
public class RestWebApplicationInitializer implements WebApplicationInitializer {

  public void onStartup(ServletContext container) throws ServletException {

    // Create the 'root' Spring application context
    AnnotationConfigWebApplicationContext rootContext =
      new AnnotationConfigWebApplicationContext();
    rootContext.register(ApplicationConfig.class);

    // Manage the life cycle of the root application context
    container.addListener(new ContextLoaderListener(rootContext));

    // Register and map the dispatcher servlet
    DispatcherServlet servlet = new RepositoryRestExporterServlet();
    ServletRegistration.Dynamic dispatcher = container.addServlet("dispatcher", servlet);
    dispatcher.setLoadOnStartup(1);
    dispatcher.addMapping("/");
```

```
  }
}
```

First, we set up an `AnnotationConfigWebApplicationContext` and register the `ApplicationConfig` JavaConfig class to be used as a Spring configuration later on. We register that `ApplicationContext` wrapped inside `ContextLoaderListener` to the actual `ServletContext`. The context will invoke the listener later, which will cause the `ApplicationContext` to be bootstrapped in turn. The code so far is pretty much the equivalent of registering a `ContextLoaderListener` inside a *web.xml* file, pointing it to an XML config file, except that we don't have to deal with XML and String-based locations but rather type-safe references to configuration. The `ApplicationContext` configured will now bootstrap an embedded database, the JPA infrastructure including a transaction manager, and enable the repositories eventually. This process has already been covered in "Bootstrapping the Sample Code" on page 44.

Right after that, we declare a `RepositoryRestExporterServlet`, which actually cares about the tricky parts. It registers quite a bit of Spring MVC infrastructure components and inspects the root application context for Spring Data repository instances. It will expose HTTP resources for each of these repositories as long as they implement `CrudRepository`. This is currently a limitation, which will be removed in later versions of the module. We map the servlet to the servlet root so that the application will be available via `http://localhost:8080` for now.

Interacting with the REST Exporter

Now that we have bootstrapped the application, let's see what we can actually do with it. We will use the command-line tool curl to interact with the system, as it provides a convenient way to trigger HTTP requests and displays responses in a way that we can show here in the book nicely. However, you can, of course, use any other client capable of triggering HTTP requests: command-line tools (like wget on Windows) or simply your web browser of choice. Note that the latter will only allow you to trigger `GET` requests through the URL bar. If you'd like to follow along with the more advanced requests (`POST`, `PUT`, `DELETE`), we recommend a browser plug-in like the Dev HTTP Client for Google Chrome. Similar tools are available for other browsers as well.

Let's trigger some requests to the application, as shown in Example 10-3. All we know right now is that we've deployed it to listen to *http://localhost:8080*, so let's see what this resource actually provides.

Example 10-3. Triggering the initial request using curl

```
$ curl -v http://localhost:8080

* About to connect() to localhost port 8080 (#0)
*   Trying ::1... connected
* Connected to localhost (::1) port 8080 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.21.4 (universal-apple-darwin11.0) libcurl/7.21.4 OpenSSL/0.9.8r
```

```
    zlib/1.2.5
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Length: 242
< Content-Type: application/json
< Server: Jetty(8.1.5.v20120716)
<
{
  "links" : [ {
    "rel" : "product",
    "href" : "http://localhost:8080/product"
  }, {
    "rel" : "order",
    "href" : "http://localhost:8080/order"
  }, {
    "rel" : "customer",
    "href" : "http://localhost:8080/customer"
  } ]
}
* Connection #0 to host localhost left intact
* Closing connection #0
```

The first thing to notice here is that we triggered the `curl` command with the `-v` flag. This flag activates verbose output, listing all the request and response headers alongside the actual response data. We see that the server returns data of the content type `application/json` by default. The actual response body contains a set of links we can follow to explore the application. Each of the links provided is actually derived from a Spring Data repository available in the `ApplicationContext`. We have a `CustomerRepository`, a `ProductRepository`, and an `OrderRepository`, so the relation type (`rel`) attributes are `customer`, `product`, and `order` (the first part of the repository name beginning with a lowercase character). The resource URIs are derived using that default as well. To customize this behavior, you can annotate the repository interface with `@RestResource`, which allows you to explicitly define `path` (the part of the URI) as well as the `rel` (the relation type).

Links

The representation of a link is usually derived from the link element defined in the Atom RFC. It basically consists of two attributes: a relation type (`rel`) and a hypertext reference (`href`) . The former defines the actual semantics of the link (and thus has to be documented or standardized), whereas the latter is actually opaque to the client. A client will usually inspect a link's response body for relation types and follow the links with relation types it is interested in. So basically the client knows it will find all orders behind links with a `rel` of `order`. The actual URI is not relevant. This structure results in decoupling of the client and the server, as the latter tells the client where to go. This is especially useful in case a URI changes or the server actually wants to point the client to a different machine to load-balance requests.

Let's move on to inspecting the products available in the system. We know that the products are exposed via the relation type product; thus, we follow the link with that `rel`.

Accessing Products

Example 10-4 demonstrates how to access all products available in the system.

Example 10-4. Accessing products

```
$ curl http://localhost:8080/product

{ "content" : [ {
    "price" : 499.00,
    "description" : "Apple tablet device",
    "name" : "iPad",
    "links" : [ {
      "rel" : "self",
      "href" : "http://localhost:8080/product/1"
    } ],
    "attributes" : {
      "connector" : "socket"
    }
  }, … , {
    "price" : 49.00,
    "description" : "Dock for iPhone/iPad",
    "name" : "Dock",
    "links" : [ {
      "rel" : "self",
      "href" : "http://localhost:8080/product/3"
    } ],
    "attributes" : {
      "connector" : "plug"
    }
  } ],
  "links" : [ {
    "rel" : "product.search",
    "href" : "http://localhost:8080/product/search"
  } ]
}
```

Triggering the request to access all products returns a JSON representation containing two major fields. The `content` field consists of a collection of all available products rendered directly into the response. The individual elements contain the serialized properties of the `Product` class as well as an artificial `links` container. This container carries a single link with a relation type of `self`. The `self` type usually acts as a kind of identifier, as it points to the resource itself. So we can access the iPad product directly by following the link with the relation type `self` inside its representation (see Example 10-5).

Example 10-5. Accessing a single product

```
$ curl http://localhost:8080/product/1

{ "price" : 499.00,
  "description" : "Apple tablet device",
  "name" : "iPad",
  "links" : [ {
    "rel" : "self",
    "href" : "http://localhost:8080/product/1"
  } ],
  "attributes" : {
    "connector" : "socket"
  }
}
```

To update a product, you simply issue a PUT request to the resource providing the new content, as shown in Example 10-6.

Example 10-6. Updating a product

```
$ curl -v -X PUT -H "Content-Type: application/json" \
       -d '{ "price" : 469.00, \
             "name" : "Apple iPad" }' \
       http://localhost:8080/spring-data-book-rest/product/1

* About to connect() to localhost port 8080 (#0)
*   Trying ::1...
* connected
* Connected to localhost (::1) port 8080 (#0)
> PUT /spring-data-book-rest/product/1 HTTP/1.1
> User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r
    zlib/1.2.5
> Host: localhost:8080
> Accept: */*
> Content-Type: application/json
> Content-Length: 82
>
* upload completely sent off: 82 out of 82 bytes
< HTTP/1.1 204 No Content
< Server: Apache-Coyote/1.1
< Date: Fri, 31 Aug 2012 10:23:58 GMT
```

We set the HTTP method to `PUT` using the `-X` parameter and provide a `Content-Type` header to indicate we're sending JSON. We submit an updated `price` and `name` attribute provided through the `-d` parameter. The server returns a `204 No Content` to indicate that the request was successful. Triggering another `GET` request to the product's URI returns the updated content, as shown in Example 10-7.

Example 10-7. The updated product

```
$ curl http://localhost:8080/spring-data-book-rest/product/1

{ "links" : [ {
    "rel" : "self",
    "href" : "http://localhost:8080/spring-data-book-rest"
  } ],
  "price" : 469.00,
  "description" : "Apple tablet device",
  "name" : "Apple iPad",
  "attributes" : {
    "connector" : "socket"
  }
}
```

The JSON representation of the collection resource also contained a `links` attribute, which points us to a generic resource that will allow us to explore the query methods exposed by the repository. The convention is using the relation type of the collection resource (in our case, `product`) extended by `.search`. Let's follow this link and see what searches we can actually execute—see Example 10-8.

Example 10-8. Accessing available searches for products

```
$ curl http://localhost:8080/product/search

{ "links" : [ {
    "rel" : "product.findByDescriptionContaining",
    "href" : "http://localhost:8080/product/search/findByDescriptionContaining"
  }, {
    "rel" : "product.findByAttributeAndValue",
    "href" : "http://localhost:8080/product/search/findByAttributeAndValue"
  } ]
}
```

As you can see, the repository exporter exposes a resource for each of the query methods declared in the `ProductRepository` interface. The relation type pattern is again based on the relation type of the resource extended by the query method name, but we can customize it using `@RestResource` on the query method. Since the JVM unfortunately doesn't support deriving the parameter names from interface methods, we have to annotate the method parameters of our query methods with `@Param` and use named parameters in our manual query method definition for `findByAttributeAndValue(...)`. See Example 10-9.

Example 10-9. The PersonRepository interface

```
public interface ProductRepository extends CrudRepository<Product, Long> {

  Page<Product> findByDescriptionContaining(
      @Param("description") String description, Pageable pageable);

  @Query("select p from Product p where p.attributes[:attribute] = :value")
  List<Product> findByAttributeAndValue(
```

```
      @Param("attribute") String attribute, @Param("value") String value);
}
```

We can now trigger the second query method by following the `product.findByAttributeAndValue` link and invoking a `GET` request handing the matching parameters to the server. Let's search for products that have a connector plug, as shown in Example 10-10.

Example 10-10. Searching for products with a connector attribute of value plug

```
$ curl http://localhost:8080/product/search/findByAttributeAndValue?attribute=connector\
      /&value=plug

{ "results" : [ {
    "price" : 49.00,
    "description" : "Dock for iPhone/iPad",
    "name" : "Dock",
    "_links" : [ {
      "rel" : "self",
      "href" : "http://localhost:8080/product/3"
    } ],
    "attributes" : {
      "connector" : "plug"
    }
  } ],
  "links" : [ … ]
}
```

Accessing Customers

Now that we've seen how to navigate through the products available and how to execute the finder methods exposed by the repository interface, let's switch gears and have a look at the customers registered in the system. Our original request to *http://localhost:8080* exposed a `customer` link (see Example 10-3). Let's follow that link in Example 10-11 and see what customers we find.

Example 10-11. Accessing customers (1 of 2)

```
$ curl -v http://localhost:8080/customer

* About to connect() to localhost port 8080 (#0)
*   Trying ::1...
* connected
* Connected to localhost (::1) port 8080 (#0)
> GET /customer HTTP/1.1
> User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r
    zlib/1.2.5
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 500 Could not write JSON: No serializer found for class
  com.oreilly.springdata.rest.core.EmailAddress and no properties
  discovered to create BeanSerializer …
< Content-Type: text/html;charset=ISO-8859-1
```

```
< Cache-Control: must-revalidate,no-cache,no-store
< Content-Length: 16607
< Server: Jetty(8.1.5.v20120716)
```

Yikes, that doesn't look too good. We're getting a `500 Server Error` response, indicating that something went wrong with processing the request. Your terminal output is probably even more verbose, but the important lines are listed in Example 10-11 right underneath the HTTP status code. Jackson (the JSON marshalling technology used by Spring Data REST) seems to choke on serializing the `EmailAddress` value object. This is due to the fact that we don't expose any getters or setters, which Jackson uses to discover properties to be rendered into the response.

Actually, we don't even want to render the `EmailAddress` as an embedded object but rather as a plain `String` value. We can achieve this by customizing the rendering using the `@JsonSerialize` annotation provided by Jackson. We configure its using attribute to the predefined `ToStringSerializer.class`, which will simply render the object by calling the `toString()` method on the object.

All right, let's give it another try (Example 10-12).

Example 10-12. Accessing customers (2 of 2)

```
$ curl -v http://localhost:8080/customer

* About to connect() to localhost port 8080 (#0)
*   Trying ::1...
* connected
* Connected to localhost (::1) port 8080 (#0)
> GET /customer HTTP/1.1
> User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r
    zlib/1.2.5
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 500 Could not write JSON: Infinite recursion (StackOverflowError)
  (through reference chain: com.oreilly.springdata.rest.core.Address["copy"]
  ->com.oreilly.springdata.rest.core.Address["copy"]…
< Content-Type: text/html;charset=ISO-8859-1
< Cache-Control: must-revalidate,no-cache,no-store
< Content-Length: 622972
< Server: Jetty(8.1.5.v20120716)
```

Well, it's not that much better, but at least we seem to get one step further. This time the Jackson renderer complains about the `Address` class exposing a `copy` property, which in turn causes a recursion. The reason for this issue is that the `getCopy()` method of `Address` class follows the Java bean property semantics but is not a getter method in the classic sense. Rather, it returns a copy of the `Address` object to allow us to easily create a clone of an `Address` instance and assign it to an `Order` to shield against changes to the `Customer`'s `Address` leaking into an already existing `Order` (see Example 4-7). So we have two options here: we could either rename the method to not match the Java bean property convention or add an annotation to tell Jackson to simply ignore the

property. We'll choose the latter for now, as we don't want to get into refactoring the client code. Thus, we use the `@JsonIgnore` annotation to exclude the `copy` property from rendering, as shown in Example 10-13.

Example 10-13. Excluding the copy property of the Address class from rendering

```
@Entity
public class Address extends AbstractEntity {

  private String street, city, country;

  …

  @JsonIgnore
  public Address getCopy() { … }
}
```

Having made this change, let's restart the server and invoke the request again (Example 10-14).

Example 10-14. Accessing customers after marshalling tweaks

```
$ curl http://localhost:8080/customer

{ "results" : [ {
    "links" : [ {
      "rel" : "self",
      "href" : "http://localhost:8080/customer/1"
    } ],
    "lastname" : "Matthews",
    "emailAddress" : "dave@dmband.com",
    "firstname" : "Dave",
    "addresses" : [ {
      "id" : 1,
      "street" : "27 Broadway",
      "city" : "New York",
      "country" : "United States"
    }, {
      "id" : 2,
      "street" : "27 Broadway",
      "city" : "New York",
      "country" : "United States"
    } ]
  }, … ],
  "links" : [ {
    "rel" : "customer.search",
    "href" : "http://localhost:8080/customer/search"
  } ],
  "page" {
    "number" : 1,
    "size" : 20,
    "totalPages" : 1,
    "totalElements" : 3
  }
}
```

As you can see, the entities can now be rendered correctly. We also find the expected links section to point us to the available query methods for customers. What's new, though, is that we have an additional `page` attribute set in the returned JSON. It contains the current page number (`number`), the requested page size (`size`, defaulted to 20 here), the total number of pages available (`totalPages`), and the overall total number of elements available (`totalElements`).

These attributes appear due to the fact that our `CustomerRepository` extends `PagingAndSortingRepository` and thus allows accessing all customers on a page-by-page basis. For more details on that, have a look at "Defining Repositories" on page 19. This allows us to restrict the number of customers to be returned for the collection resource by using page and limit parameters when triggering the request. As we have three customers present, let's request an artificial page size of one customer, as shown in Example 10-15.

Example 10-15. Accessing the first page of customers

```
$ curl http://localhost:8080/customer?limit=1

{ "content" : [ … ],
  "links" : [ {
    "rel" : "customer.next",
    "href" : "http://localhost:8080/customer?page=2&limit=1"
  }, {
    "rel" : "customer.search",
    "href" : "http://localhost:8080/customer/search"
  } ],
  "page" : {
    "number" : 1,
    "size" : 1,
    "totalPages" : 3,
    "totalElements" : 3
  }
}
```

Note how the metainformation provided alongside the single result changed. The `totalPages` field now reflects three pages being available due to our selecting a page size of one. Even better, the server indicates that we can navigate the customers to the next page by following the `customer.next` link. It already includes the request parameters needed to request the second page, so the client doesn't need to construct the URI manually. Let's follow that link and see how the metadata changes while navigating through the collection (Example 10-16).

Example 10-16. Accessing the second page of customers

```
$ curl http://localhost:8080/customer?page=2\&limit=1

{ "content" : [ … ],
  "links" : [ {
    "rel" : "customer.prev",
    "href" : "http://localhost:8080/customer?page=1&limit=1"
```

```
  }, {
    "rel" : "customer.next",
    "href" : "http://localhost:8080/customer?page=3&limit=1"
  }, {
    "rel" : "customer.search",
    "href" : "http://localhost:8080/customer/search"
  } ],
  "page" : {
    "number" : 2,
    "size" : 1,
    "totalPages" : 3,
    "totalElements" : 3
  }
}
```

Note that you might have to escape the `&` when pasting the URI into the console shown in Example 10-16. If you're working with a dedicated HTTP client, escaping is not necessary.

Besides the actual content returned, note how the `number` attribute reflects our move to page two. Beyond that, the server detects that there is now a previous page available and offers to navigate to it through the `customer.prev` link. Following the `customer.next` link a second time would result in the next representation not listing the `customer.next` link anymore, as we have reached the end of the available pages.

Accessing Orders

The final root link relation to explore is `order`. As the name suggests, it allows us to access the `Order`s available in the system. The repository interface backing the resource is `OrderRepository`. Let's access the resource and see what gets returned by the server (Example 10-17).

Example 10-17. Accessing orders

```
$ curl http://localhost:8080/order

{ "content" : [ {
    "billingAddress" : {
      "id" : 2,
      "street" : "27 Broadway",
      "city" : "New York",
      "country" : "United States"
    },
    "shippingAddress" : {
      "id" : 2,
      "street" : "27 Broadway",
      "city" : "New York",
      "country" : "United States"
    },
    "lineItems" : [ … ]
```

```
    "links" : [ {
      "rel" : "order.Order.customer",
      "href" : "http://localhost:8080/order/1/customer"
    }, {
      "rel" : "self",
      "href" : "http://localhost:8080/order/1"
    } ],
  } ],
  "links" : [ {
    "rel" : "order.search",
    "href" : "http://localhost:8080/order/search"
  } ],
  "page" : {
    "number" : 1,
    "size" : 20,
    "totalPages" : 1,
    "totalElements" : 1
  }
}
```

The response contains a lot of well-known patterns we already have discussed: a link to point to the exposed query methods of the `OrderRepository`, and the nested content field, which contains serialized `Order` objects, inlined `Address` objects, and `LineItem`s. Also, we see the pagination metadata due to `OrderRepository` implementing `PagingAndSortingRepository`.

The new thing to notice here is that the `Customer` instance held in the `Order` object is not inline but instead pointed to by a link. This is because `Customer`s are managed by a Spring Data repository. Thus, they are exposed as subordinate resources of the `Order` to allow for manipulating the assignment. Let's follow that link and access the `Customer` who triggered the `Order`, as shown in Example 10-18.

Example 10-18. Accessing the Customer who placed the Order

```
$ curl http://localhost:8080/order/1/customer

{ "links" : [ {
    "rel" : "order.Order.customer.Customer",
    "href" : "http://localhost:8080/order/1/customer"
  }, {
    "rel" : "self",
    "href" : "http://localhost:8080/customer/1"
  } ],
  "emailAddress" : "dave@dmband.com",
  "lastname" : "Matthews",
  "firstname" : "Dave",
  "addresses" : [ {
    "street" : "27 Broadway",
    "city" : "New York",
    "country" : "United States"
  }, {
    "street" : "27 Broadway",
    "city" : "New York",
```

```
    "country" : "United States"
  } ]
}
```

This call returns the detailed information of the linked `Customer` and provides two links. The one with the `order.Order.customer.Customer` relation type points to the `Customer` connection resource, whereas the `self` link points to the actual `Customer` resource. What's the difference here? The former represents the assignment of the `Customer` to the order. We can alter this assignment by issuing a `PUT` request to the URI, and we could unassign it by triggering a `DELETE` request to it. In our case, the `DELETE` call will result in a `405 Method not allowed`, as the JPA mapping requires a `Customer` to be mapped via the `optional = false` flag in the `@ManyToOne` annotation of the `customer` property. If the relationship is optional, a `DELETE` request will just work fine.

Assume we discovered that it's actually not Dave who placed the `Order` initially, but Carter. How do we update the assignment? First, the HTTP method of choice is `PUT`, as we already know the URI of the resource to manipulate. It wouldn't really make sense to put actual data to the server, since all we'd like to do is tell the server "this existing `Customer` is the issuer of the `Order`." Because the `Customer` is identified through its URI, we're going to `PUT` it to the server, setting the `Content-Type` request header to `text/uri-list` so that the server knows what we send. See Example 10-19.

Example 10-19. Changing the Customer who placed an Order

```
$ curl -v -X PUT -H "Content-Type: text/uri-list" \
       -d "http://localhost:8080/customer/2" http://localhost:8080/order/1/customer

* About to connect() to localhost port 8080 (#0)
*   Trying ::1...
* connected
* Connected to localhost (::1) port 8080 (#0)
> PUT /order/1/customer HTTP/1.1
> User-Agent: curl/7.24.0 (x86_64-apple-darwin12.0) libcurl/7.24.0 OpenSSL/0.9.8r zlib/1.2.5
> Host: localhost:8080
> Accept: */*
> Content-Type: text/uri-list
> Content-Length: 32
>
* upload completely sent off: 32 out of 32 bytes
< HTTP/1.1 204 No Content
< Content-Length: 0
< Content-Type: application/octet-stream
< Server: Jetty(8.1.5.v20120716)
```

`text/uri-list` is a standardized media type to define the format of one or more URIs being transferred. Note that we get a `204 No Content` from the server, indicating that it has accepted the request and completed the reassignment.

PART V

Big Data

CHAPTER 11
Spring for Apache Hadoop

Apache Hadoop is an open source project that originated in Yahoo! as a central component in the development of a new web search engine. Hadoop's architecture is based on the architecture Google developed to implement its own closed source web search engine, as described in two research publications that you can find here and here. The Hadoop architecture consists of two major components: a distributed filesystem and a distributed data processing engine that run on a large cluster of commodity servers. The Hadoop Distributed File System (HDFS) is responsible for storing and replicating data reliably across the cluster. Hadoop MapReduce is responsible for providing the programming model and runtime that is optimized to execute the code close to where the data is stored. The colocation of code and data on the same physical node is one of the key techniques used to minimize the time required to process large amounts (up to petabytes) of data.

While Apache Hadoop originated out of a need to implement a web search engine, it is a general-purpose platform that can be used for a wide variety of large-scale data processing tasks. The combination of open source software, low cost of commodity servers, and the real-world benefits that result from analyzing large amounts of new unstructured data sources (e.g., tweets, logfiles, telemetry) has positioned Hadoop to be a de facto standard for enterprises looking to implement big data solutions.

In this chapter, we start by introducing the "Hello world" application of Hadoop, wordcount. The wordcount application is written using the Hadoop MapReduce API. It reads text files as input and creates an output file with the total count of each word it has read. We will first show how a Hadoop application is traditionally developed and executed using command-line tools and then show how this application can be developed as a standard Java application and configured using dependency injection. To copy the input text files into HDFS and the resulting output file out of HDFS, we use Spring for Apache Hadoop's HDFS scripting features.

Challenges Developing with Hadoop

There are several challenges you will face when developing Hadoop applications. The first challenge is the installation of a Hadoop cluster. While outside the scope of this book, creating and managing a Hadoop cluster takes a significant investment of time, as well as expertise. The good news is that many companies are actively working on this front, and offerings such as Amazon's Elastic Map Reduce let you get your feet wet using Hadoop without significant upfront costs. The second challenge is that, very often, developing a Hadoop application does not consist solely of writing a single MapReduce, Pig, or Hive job, but rather it requires you to develop a complete data processing pipeline. This data pipeline consists of the following steps:

1. Collecting the raw data from many remote machines or devices.
2. Loading data into HDFS, often a continuous process from diverse sources (e.g., application logs), and event streams.
3. Performing real-time analysis on the data as it moves through the system and is loaded into HDFS.
4. Data cleansing and transformation of the raw data in order to prepare it for analysis.
5. Selecting a framework and programming model to write data analysis jobs.
6. Coordinating the execution of many data analysis jobs (e.g., workflow). Each individual job represents a step to create the final analysis results.
7. Exporting final analysis results from HDFS into structured data stores, such as a relational database or NoSQL databases like MongoDB or Redis, for presentation or further analysis.

Spring for Apache Hadoop along with two other Spring projects, Spring Integration and Spring Batch, provide the basis for creating a complete data pipeline solution that has a consistent configuration and programming model. While this topic is covered in Chapter 13, in this chapter we must start from the basics: how to interact with HDFS and MapReduce, which in itself provides its own set of challenges.

Command-line tools are currently promoted in Hadoop documentation and training classes as the primary way to interact with HDFS and execute data analysis jobs. This feels like the logical equivalent of using SQL*Plus to interact with Oracle. Using command-line tools can lead you down a path where your application becomes a loosely organized collection of bash, Perl, Python, or Ruby scripts. Command-line tools also require you to create ad-hoc conventions to parameterize the application for different environments and to pass information from one processing step to another. There should be an easy way to interact with Hadoop programmatically, as you would with any other filesystem or data access technology.

Spring for Apache Hadoop aims to simplify creating Hadoop-based applications in Java. It builds upon the Spring Framework to provide structure when you are writing Hadoop applications. It uses the familiar Spring-based configuration model that lets

you take advantage of the powerful configuration features of the Spring container, such as property placeholder replacement and portable data access exception hierarchies. This enables you to write Hadoop applications in the same style as you would write other Spring-based applications.

Hello World

The classic introduction to programming with Hadoop MapReduce is the wordcount example. This application counts the frequency of words in text files. While this sounds simple to do using Unix command-line utilities such as `sed`, `awk`, or `wc`, what compels us to use Hadoop for this is how well this problem can scale up to match Hadoop's distributed nature. Unix command-line utilities can scale to many megabytes or perhaps a few gigabytes of data. However, they are a single process and limited by the disk transfer rates of a single machine, which are on the order of 100 MB/s. Reading a 1 TB file would take about two and a half hours. Using Hadoop, you can scale up to hundreds of gigabytes, terabytes, or even petabytes of data by distributing the data across the HDFS cluster. A 1 TB dataset spread across 100 machines would reduce the read time to under two minutes. HDFS stores parts of a file across many nodes in the Hadoop cluster. The MapReduce code that represents the logic to perform on the data is sent to the nodes where the data resides, executing close to the data in order to increase the I/O bandwidth and reduce latency of the overall job. This stage is the "Map" stage in MapReduce. To join the partial results from each node together, a single node in the cluster is used to "Reduce" the partial results into a final set of data. In the case of the wordcount example, the word counts accumulated on individual machines are combined into the final list of word frequencies.

The fun part of running wordcount is selecting some sample text to use as input. While it was surely not the intention of the original authors, Project Gutenberg provides an easily accessible means of downloading large amounts of text in the form of public domain books. Project Gutenberg is an effort to digitize the full text of public domain books and has over 39,000 books currently available. You can browse the project website and download a few classic texts using *wget*. In Example 11-1, we are executing the command in the directory */tmp/gutenberg/download*.

Example 11-1. Using wget to download books for wordcount

```
wget -U firefox  http://www.gutenberg.org/ebooks/4363.txt.utf-8
```

Now we need to get this data into HDFS using a HDFS shell command.

Before running the shell command, we need an installation of Hadoop. A good guide to setting up your own Hadoop cluster on a single machine is described in Michael Noll's excellent online tutorial.

We invoke an HDFS shell command by calling the `hadoop` command located in the *bin* directory of the Hadoop distribution. The Hadoop command-line argument `dfs` lets you work with HDFS and in turn is followed by traditional file commands and arguments, such as `cat` or `chmod`. The command to copy files from the local filesystem into HDFS is `copyFromLocal`, as shown in Example 11-2.

Example 11-2. Copying local files into HDFS

```
hadoop dfs -copyFromLocal /tmp/gutenberg/download /user/gutenberg/input
```

To check if the files were stored in HDFS, use the `ls` command, as shown in Example 11-3.

Example 11-3. Browsing files in HDFS

```
hadoop dfs -ls /user/gutenberg/input
```

To run the wordcount application, we use the example jar file that ships as part of the Hadoop distribution. The arguments for the application is the name of the application to run—in this case, wordcount—followed by the HDFS input directory and output directory, as shown in Example 11-4.

Example 11-4. Running wordcount using the Hadoop command-line utility

```
hadoop jar hadoop-examples-1.0.1.jar wordcount /user/gutenberg/input
  /user/gutenberg/output
```

After issuing this command, Hadoop will churn for a while, and the results will be placed in the directory */user/gutenberg/output*. You can view the output in HDFS using the command in Example 11-5.

Example 11-5. View the output of wordcount in HDFS

```
hadoop dfs -cat /user/gutenberg/output/part-r-00000
```

Depending on how many input files you have, there may be more than one output file. By default, output filenames follow the scheme shown in Example 11-5 with the last set of numbers incrementing for each additional file that is output. To copy the results from HDFS onto the local filesystem, use the command in Example 11-6.

Example 11-6. View the output of wordcount in HDFS

```
hadoop dfs -getmerge /user/gutenberg/output /tmp/gutenberg/output/wordcount.txt
```

If there are multiple output files in HDFS, the `getmerge` option merges them all into a single file when copying the data out of HDFS to the local filesystem. Listing the file contents shows the words in alphabetical order followed by the number of times they appeared in the file. The superfluous-looking quotes are an artifact of the implementation of the MapReduce code that tokenized the words. Sample output of the wordcount application output is shown in Example 11-7.

Example 11-7. Partial listing of the wordcount output file

```
> cat /tmp/gutenberg/output/wordcount.txt
A 2
"AWAY 1
"Ah, 1
"And 2
"Another 1
…
"By 2
"Catholicism" 1
"Cease 1
"Cheers 1
…
```

In the next section, we will peek under the covers to see what the sample application that is shipped as part of the Hadoop distribution is doing to submit a job to Hadoop. This will help you understand what's needed to develop and run your own application.

Hello World Revealed

There are a few things going on behind the scenes that are important to know if you want to develop and run your own MapReduce applications and not just the ones that come out of the box. To get an understanding of how the example applications work, start off by looking in the *META-INF/manifest.mf* file in *hadoop-examples.jar*. The manifest lists `org.apache.hadoop.examples.ExampleDriver` as the main class for Java to run. `ExampleDriver` is responsible for associating the first command-line argument, `wordcount`, with the Java class `org.apache.hadoop.examples.Wordcount` and executing the main method of `Wordcount` using the helper class `ProgramDriver`. An abbreviated version of `ExampleDriver` is shown in Example 11-8.

Example 11-8. Main driver for the out-of-the-box wordcount application

```
public class ExampleDriver {

  public static void main(String... args){

    int exitCode = -1;
    ProgramDriver pgd = new ProgramDriver();

    try {
      pgd.addClass("wordcount", WordCount.class,
                   "A map/reduce program that counts the words in the input files.");
      pgd.addClass("randomwriter", RandomWriter.class,
                   "A map/reduce program that writes 10GB of random data per node.");

      // additional invocations of addClass excluded that associate keywords
      // with other classes

      exitCode = pgd.driver(args);
    } catch(Throwable e) {
```

```
      e.printStackTrace();
    }

    System.exit(exitCode);
  }
}
```

The `WordCount` class also has a main method that gets invoked not directly by the JVM when starting, but when the `driver` method of `ProgramDriver` is invoked. The `WordCount` class is shown in Example 11-9.

Example 11-9. The wordcount main method invoked by the ProgramDriver

```
public class WordCount {

  public static void main(String... args) throws Exception {

    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();

    if (otherArgs.length != 2) {
      System.err.println("Usage: wordcount <in> <out>");
      System.exit(2);
    }

    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

This gets at the heart of what you have to know in order to configure and execute your own MapReduce applications. The necessary steps are: create a new Hadoop `Configuration` object, create a `Job` object, set several job properties, and then run the job using the method `waitforCompletion(…)`. The `Mapper` and `Reducer` classes form the core of the logic to write when creating your own application.

While they have rather generic names, `TokenizerMapper` and `IntSumReducer` are static inner classes of the `WordCount` class and are responsible for counting the words and summing the total counts. They're demonstrated in Example 11-10.

Example 11-10. The Mapper and Reducer for the out-of-the-box wordcount application

```
public class WordCount {

  public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable>{
```

```
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context)
      throws IOException, InterruptedException {

      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer extends Reducer<Text,IntWritable,Text,IntWritable> {

    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {

      int sum = 0;

      for (IntWritable val : values) {
        sum += val.get();
      }

      result.set(sum);
      context.write(key, result);
    }
  }

  // … main method as shown before
}
```

Since there are many out-of-the-box examples included in the Hadoop distribution, the `ProgramDriver` utility helps to specify which Hadoop job to run based off the first command-line argument. You can also run the `WordCount` application as a standard Java main application without using the `ProgramDriver` utility. A few minor modifications related to command-line argument handling are all that you need. The modified `WordCount` class is shown in Example 11-11.

Example 11-11. A standalone wordcount main application

```
public class WordCount {

  // ... TokenizerMapper shown before
  // ... IntSumReducer shown before

  public static void main(String[] args) throws Exception {

    Configuration conf = new Configuration();
```

```
    if (args.length != 2) {
      System.err.println("Usage: <in> <out>");
      System.exit(2);
    }

    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

The sample application for this section is located in the directory *./hadoop/wordcount*. A Maven build file for `WordCount` application is also provided; this lets you run the `WordCount` application as part of a regular Java application, independent of using the Hadoop command-line utility. Using Maven to build your application and run a standard Java main application is the first step toward treating the development and deployment of Hadoop applications as regular Java applications, versus one that requires a separate Hadoop command-line utility to execute. The Maven build uses the Appassembler plug-in to generate Unix and Windows start scripts and to collect all the required dependencies into a local *lib* directory that is used as the classpath by the generated scripts.

To rerun the previous `WordCount` example using the same output direction, we must first delete the existing files and directory, as Hadoop does not allow you to write to a preexisting directory. The command `rmr` in the HDFS shell achieves this goal, as you can see in Example 11-12.

Example 11-12. Removing a directory and its contents in HDFS

```
hadoop dfs -rmr /user/gutenberg/output
```

To build the application, run Appassembler's assemble target and then run the generated wordcount shell script (Example 11-13).

Example 11-13. Building, running, and viewing the output of the standalone wordcount application

```
$ cd hadoop/wordcount
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/wordcount hdfs://localhost:9000/user/gutenberg/input
     hdfs://localhost:9000/user/gutenberg/output

INFO: Total input paths to process : 1
INFO: Running job: job_local_0001
…
```

```
INFO:     Map output records=65692

$ hadoop dfs -cat /user/gutenberg/output/part-r-00000
"A 2
"AWAY 1
"Ah, 1
"And 2
"Another 1
"Are 2
"BIG 1
…
```

One difference to using the Hadoop command line is that the directories in HDFS need to be prefixed with the URL scheme `hdfs://` along with the hostname and port of the namenode. This is required because the Hadoop command line sets environment variables recognized by the Hadoop `Configuration` class that prepend this information to the paths passed in as command-line arguments. Note that other URL schemes for Hadoop are available. The `webhdfs` scheme is very useful because it provides an HTTP-based protocol to communicate with HDFS that does not require the client (our application) and the HDFS server to use the exact same version (down to the minor point release) of the HDFS libraries.

Hello World Using Spring for Apache Hadoop

If you have been reading straight through this chapter, you may well be wondering, what does all this have to do with Spring? In this section, we start to show the features that Spring for Apache Hadoop provides to help you structure, configure, and run Hadoop applications. The first feature we will examine is how to use Spring to configure and run Hadoop jobs so that you can externalize application parameters in separate configuration files. This lets your application easily adapt to running in different environments—such as development, QA, and production—without requiring any code changes.

Using Spring to configure and run Hadoop jobs lets you take advantage of Spring's rich application configuration features, such as property placeholders, in the same way you use Spring to configure and run other applications. The additional effort to set up a Spring application context might not seem worth it for such a simple application as wordcount, but it is rare that you'll build such simple applications. Applications will typically involve chaining together several HDFS operations and MapReduce jobs (or equivalent Pig and Hive scripts). Also, as mentioned in "Challenges Developing with Hadoop" on page 176, there are many other development activities that you need to consider in creating a complete data pipeline solution. Using Spring for Apache Hadoop as a basis for developing Hadoop applications gives us a foundation to build upon and to reuse components as our application complexity grows.

Let's start by running the version of `WordCount` developed in the previous section inside of the Spring container. We use the XML namespace for Hadoop to declare the location

of the namenode and the minimal amount of information required to define a `org.apache.hadoop.mapreduce.Job` instance. (See Example 11-14.)

Example 11-14. Declaring a Hadoop job using Spring's Hadoop namespace

```
<configuration>
  fs.default.name=hdfs://localhost:9000
</configuration>

<job id="wordcountJob"
     input-path="/user/gutenberg/input"
     output-path="/user/gutenberg/output"
     mapper="org.apache.hadoop.examples.WordCount.TokenizerMapper"
     reducer="org.apache.hadoop.examples.WordCount.IntSumReducer"/>

<job-runner id="runner" job="wordcountJob" run-at-startup="true"/>
```

This configuration will create a singleton instance of an `org.apache.hadoop.mapreduce.Job` managed by the Spring container. Some of the properties that were set on the job class programmatically in the previous example can be inferred from the class signature of the `Mapper` and `Reducer`. Spring can determine that `outputKeyClass` is of the type `org.apache.hadoop.io.Text` and that `outputValueClass` is of type `org.apache.hadoop.io.IntWritable`, so we do not need to set these properties explicitly. There are many other job properties you can set that are similar to the Hadoop command-line options (e.g., combiner, input format, output format, and general job key/value properties). Use XML schema autocompletion in Eclipse or another editor to see the various options available, and also refer to the Spring for Apache Hadoop reference documentation for more information. Right now, the namenode location, input, and output paths are hardcoded. We will extract them to an external property file shortly.

Similar to using the Hadoop command line to run a job, we don't need to specify the URL scheme and namenode host and port when specifying the input and output paths. The `<configuration/>` element defines the default URL scheme and namenode information. If you wanted to use the `webhdfs` protocol, then you simply need to set the value of the key `fs.default.name` to `webhdfs://localhost`. You can also specify values for other Hadoop configuration keys, such as `dfs.permissions`, `hadoop.job.ugi`, `mapred.job.tracker`, and `dfs.datanode.address`.

To launch a MapReduce job when a Spring `ApplicationContext` is created, use the utility class `JobRunner` to reference one or more managed `Job` objects and set the `run-at-startup` attribute to `true`. The main application class, which effectively takes the place of `org.apache.hadoop.examples.ExampleDriver`, is shown in Example 11-15. By default, the application looks in a well-known directory for the XML configuration file, but we can override this by providing a command-line argument that references another configuration location.

Example 11-15. Main driver to custom wordcount application managed by Spring

```
public class Main {

  private static final String[] CONFIGS = new String[] {
      "META-INF/spring/hadoop-context.xml" };

  public static void main(String[] args) {
    String[] res = (args != null && args.length > 0 ? args : CONFIGS);
    AbstractApplicationContext ctx = new ClassPathXmlApplicationContext(res);
    // shut down the context cleanly along with the VM
    ctx.registerShutdownHook();
  }
}
```

The sample code for this application is located in *./hadoop/wordcount-spring-basic*. You can build and run the application just like in the previous sections, as shown in Example 11-16. Be sure to remove the output files in HDFS that were created from running wordcount in previous sections.

Example 11-16. Building and running a basic Spring-based wordcount application

```
$ hadoop dfs -rmr /user/gutenberg/output
$ cd hadoop/wordcount-spring-basic
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/wordcount
```

Now that the Hadoop job is a Spring-managed object, it can be injected into any other object managed by Spring. For example, if we want to have the wordcount job launched in a web application, we can inject it into a Spring MVC controller, as shown in Example 11-17.

Example 11-17. Dependency injection of a Hadoop job in WebMVC controller

```
@Controller
public class WordController {
  private final Job mapReduceJob;

  @Autowired
  public WordService(Job mapReduceJob) {
    Assert.notNull(mapReducejob);
    this.mapReduceJob = mapReduceJob;
  }

  @RequestMapping(value = "/runjob", method = RequestMethod.POST)
  public void runJob() {
    mapReduceJob.waitForCompletion(false);
  }
}
```

To start and externalize the configuration parameters of the application, we use Spring's `property-placeholder` functionality and move key parameters to a configuration file (Example 11-18).

Example 11-18. Declaring a parameterized Hadoop job using Spring's Hadoop namespace

```
<context:property-placeholder location="hadoop-default.properties"/>

<configuration>
  fs.default.name=${hd.fs}
</configuration>

<job id="wordcountJob"
     input-path="${wordcount.input.path}"
     output-path="${wordcount.output.path}"
     mapper="org.apache.hadoop.examples.WordCount.TokenizerMapper"
     reducer="org.apache.hadoop.examples.WordCount.IntSumReducer"/>

<job-runner id="runner" job="wordcountJob" run-at-startup="true"/>
```

The variable names `hd.fs`, `wordcount.input.path`, and `wordcount.output.path` are specified in the configuration file *hadoop-default.properties*, as shown in Example 11-19.

Example 11-19. The property file, hadoop-default.properties, that parameterizes the Hadoop application for the default development environment

```
hd.fs=hdfs://localhost:9000
wordcount.input.path=/user/gutenberg/input/
wordcount.output.path=/user/gutenberg/output/
```

This file is located in the *src/main/resources* directory so that it is made available on the classpath by the build script. We also can create another configuration file, named *hadoop-qa.properties*, which will define the location of the namenode as configured in the QA environment. To run the example on the same machine, we change only the name of the output path, as shown in Chapter 11. In a real QA environment, the location of the HDFS cluster, as well as the HDFS input and output paths, would be different.

Example 11-20. The property file, hadoop-qa.properties, that parameterizes the Hadoop application for the QA environment

```
hd.fs=hdfs://localhost:9000
wordcount.input.path=/data/words/input
wordcount.output.path=/data/words/qa/output
```

To take advantage of Spring's environment support, which enables easy switching between different sets of configuration files, we change the `property-placeholder` definition to use the variable `${ENV}` in the name of the property file to load. By default, Spring will resolve variable names by searching through JVM system properties and then environment variables. We specify a default value for the variable by using the syntax `${ENV:<Default Value>}`. In Example 11-21, if the shell environment variable, `ENV`, is not set, the value `default` will be used for `${ENV}` and the property file `hadoop-default.properties` will be loaded.

Example 11-21. Referencing different configuration files for different runtime envrionments

```
<context:property-placeholder location="hadoop-${ENV:default}.properties"/>
```

To run the application from the command line for the QA environment, run the commands in Example 11-22. Notice how the shell environment variable (`ENV`) is set to `qa`.

Example 11-22. Building and running the intermediate Spring-based wordcount application in the QA environment

```
$ hadoop dfs -copyFromLocal /tmp/gutenberg/download /user/gutenberg/input
$ hadoop dfs -rmr /user/gutenberg/qa/output
$ cd hadoop/wordcount-spring-intermediate
$ mvn clean package appassembler:assemble
$ export ENV=qa
$ sh ./target/appassembler/bin/wordcount
```

As we have seen over the course of these examples, when rerunning a Hadoop application we always need to write out the results to an empty directory; otherwise, the Hadoop job will fail. In the development cycle, it's tedious to remember to do this before launching the application each time, in particular when inside the IDE. One solution is to always direct the output to a new directory name based on a timestamp (e.g., `/user/gutenberg/output/2012/6/30/14/30`) instead of a static directory name. Let's see how we can use Spring's HDFS scripting features to help us with this common task.

Scripting HDFS on the JVM

When developing Hadoop applications, you will quite often need to interact with HDFS. A common way to get started with it is to use the HDFS command-line shell. For example, here's how to get a list of the files in a directory:

```
hadoop dfs -ls /user/gutenberg/input
```

While that is sufficient for getting started, when writing Hadoop applications you often need to perform more complex chains of filesystem operations. For example, you might need to test if a directory exists; if it does, delete it, and copy in some new files. As a Java developer, you might feel that adding this functionality into bash scripts is a step backward. Surely there is a programmatic way to do this, right? Good news: there is. It is the HDFS filesystem API. The bad news is that the Hadoop filesystem API is not very easy to use. It throws checked exceptions and requires us to construct `Path` instances for many of its method arguments, making the calling structure more verbose and awkward than needed. In addition, the Hadoop filesystem API does not provide many of the higher-level methods available from the command line, such as `test` and `chmod`.

Spring for Apache Hadoop provides an intermediate ground. It provides a simple wrapper for Hadoop's `FileSystem` class that accepts `String`s instead of `Path` arguments. More importantly, it provides an `FsShell` class that mimics the functionality in the

command- line shell but is meant to be used in a programmatic way. Instead of printing out information to the console, `FsShell` methods return objects or collections that you can inspect and use programmatically. The `FsShell` class was also designed to integrate with JVM-based scripting languages, allowing you to fall back to a scripting style interaction model but with the added power of using JRuby, Jython, or Groovy instead of bash. Example 11-23 uses the `FsShell` from inside a Groovy script.

Example 11-23. Declaring a Groovy script to execute

```
<configuration>
  fs.default.name=hdfs://localhost:9000
</configuration>

<script location="org/company/basic-script.groovy"/>
```

The `<script/>` element is used to create an instance of `FsShell`, which is implicitly passed into the Groovy script under the variable name `fsh`. The Groovy script is shown in Example 11-24.

Example 11-24. Using a Groovy script to work with HDFS

```
srcDir = "/tmp/gutenberg/download/"

// use the shell (made available under variable fsh)
dir = "/user/gutenberg/input"
if (!fsh.test(dir)) {
   fsh.mkdir(dir)
   fsh.copyFromLocal(srcDir, dir)
   fsh.chmod(700, dir)
}
```

Additional options control when the script gets executed and how it is evaluated. To avoid executing the script at startup, set `run-at-startup="false"` in the `script` tag's element. To reevaluate the script in case it was changed on the filesystem, set `evaluate="IF_MODIFIED"` in the `script` tag's element.

You can also parameterize the script that is executed and pass in variables that are resolved using Spring's property placeholder functionality, as shown in Example 11-25.

Example 11-25. Configuring a parameterized Groovy script to work with HDFS

```
<context:property-placeholder location="hadoop.properties"/>

<configuration>
  fs.default.name=${hd.fs}
</configuration>

<script id="setupScript" location="copy-files.groovy">
  <property name="localSourceFile" value="${localSourceFile}"/>
  <property name="inputDir" value="${inputDir}"/>
  <property name="outputDir" value="${outputDir}"/>
</script>
```

The property file *hadoop.properties* and *copy-files.groovy* are shown in Examples 11-26 and 11-27, respectively.

Example 11-26. Property file containing HDFS scripting variables

```
hd.fs=hdfs://localhost:9000
localSourceFile=data/apache.log
inputDir=/user/gutenberg/input/word/
outputDir=
  #{T(org.springframework.data.hadoop.util.PathUtils).format('/output/%1$tY/%1$tm/%1$td')}
```

Spring for Apache Hadoop also provides a `PathUtils` class that is useful for creating time-based directory paths. Calling the static `format` method will generate a time-based path, based on the current date, that uses `java.util.Formatter`'s convention for formatting time. You can use this class inside Java but also reference it inside configuration properties by using Spring's expression langauge, SpEL. The syntax of SpEL is similar to Java, and expressions are generally just one line of code that gets evaluated. Instead of using the syntax `${...}` to reference a variable, use the syntax `#{...}` to evaluate an expression. In SpEL, the special 'T' operator is used to specify an instance of a `java.lang.Class`, and we can invoke static methods on the 'T' operator.

Example 11-27. Parameterized Groovy script to work with HDFS

```
if (!fsh.test(hdfsInputDir)) {
   fsh.mkdir(hdfsInputDir);
   fsh.copyFromLocal(localSourceFile, hdfsInputDir);
   fsh.chmod(700, hdfsInputDir)
}
if (fsh.test(hdfsOutputDir)) {
   fsh.rmr(hdfsOutputDir)
}
```

Similar to how `FsShell` makes the command-line HDFS shell operations easily available in Java or JVM scripting languages, the class `org.springframework.data.hadoop.fs.DistCp` makes the Hadoop command-line `distcp` operations easily available. The `distcp` utility is used for copying large amounts of files within a single Hadoop cluster or between Hadoop clusters, and leverages Hadoop itself to execute the copy in a distributed manner. The `distcp` variable is implicitly exposed to the script as shown in Example 11-28, which demonstrates a script embedded inside of the `<hdp:script/>` element and not in a separate file.

Example 11-28. Using distcp inside a Groovy script

```
<configuration>
  fs.default.name=hdfs://localhost:9000
</configuration>

<script language="groovy">
  distcp.copy("${src}", "${dest}")
</script>
```

In this case, the Groovy script is embedded inside the XML instead of referencing an external file. It is also important to note that we can use Spring's property placeholder functionality to reference variables such as `${src} and ${dst}` inside the script, letting you parameterize the scripts.

Combining HDFS Scripting and Job Submission

A basic Hadoop application will have some HDFS operations and some job submissions. To sequence these tasks, you can use the `pre-action` and `post-action` attributes of the `JobRunner` so that HDFS scription operations occur before and after the job submission. This is illustrated in Example 11-29.

Example 11-29. Using dependencies between beans to control execution order

```
<context:property-placeholder location="hadoop.properties"/>

<configuration>
  fs.default.name=${hd.fs}
</configuration>

<job id="wordcountJob"
     input-path="${wordcount.input.path}"
     output-path="${wordcount.output.path}"
     mapper="org.apache.hadoop.examples.WordCount.TokenizerMapper"
     reducer="org.apache.hadoop.examples.WordCount.IntSumReducer"/>

<script id="setupScript" location="copy-files.groovy">
  <property name="localSourceFile" value="${localSourceFile}"/>
  <property name="inputDir" value="${wordcount.input.path}"/>
  <property name="outputDir" value="${wordcount.output.path}"/>
</script>

<job-runner id="runner" run-at-startup="true"
            pre-action="setupScript"
            job="wordcountJob"/>
```

The `pre-action` attribute references the `setupScript` bean, which in turn references the *copy-files.groovy* script that will reset the state of the system such that this application can be run and rerun without any interaction on the command line required. Build and run the application using the commands shown in Example 11-30.

Example 11-30. Building and running the intermediate Spring-based wordcount application

```
$ hadoop dfs -rmr /user/gutenberg/output
$ cd hadoop/wordcount-hdfs-copy
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/wordcount
```

Example 11-31 shows the `JobRunner` configured to execute several HDFS scripts before and after the execution of multiple Hadoop jobs. The `JobRunner` also implements Java's `Callable` interface, which makes executing the `JobRunner` easy using Java's Executor

framework. While this approach is convenient for executing simple workflows, it only goes so far as your application becomes more complex. Treating the chain of HDFS and job operations as a first class concern is where the Hadoop-specific extensions to Spring Batch come in handy. These extensions are discussed in the section "Hadoop Workflows" on page 238.

Example 11-31. Configuring the JobRunner to execute multiple HDFS scripts and jobs

```
<job-runner id="runner"
            pre-action="setupScript1,setupScript"
            job="wordcountJob1,wordcountJob2"
            post-action="cleanupScript1,cleanupScript2"/>
```

Job Scheduling

Applications often require tasks to be scheduled and executed. The task could be sending an email or running a time-consuming end-of-day batch process. *Job schedulers* are a category of software that provides this functionality. As you might expect, there is a wide range of product offerings in this category, from the general-purpose Unix cron utility to sophisticated open source and commercial products such as Quartz, Control-M, and Autosys. Spring provides several ways to schedule jobs. They include the JDK Timer, integration with Quartz, and Spring's own TaskScheduler. In this section, we will show how you can use Quartz and Spring's TaskScheduler to schedule and execute Hadoop applications.

Scheduling MapReduce Jobs with a TaskScheduler

In this example, we will add task scheduling functionality to the application developed in the previous section, which executed an HDFS script and the wordcount MapReduce job. Spring's `TaskScheduler` and `Trigger` interfaces provides the basis for scheduling tasks that will run at some point in the future. Spring provides several implementations, with common choices being `ThreadPoolTaskScheduler` and `CronTrigger`. We can use an XML namespace, in addition to an annotation-based programming model, to configure tasks to be scheduled with a trigger.

In the configuration shown in Example 11-32, we show the use of an XML namespace that will invoke a method on any Spring-managed object—in this case, the `call` method on the `JobRunner` instance. The trigger is a cron expression that will fire every 30 seconds starting on the third second of the minute.

Example 11-32. Defining a TaskScheduler to execute HDFS scripts and MapReduce jobs

```
<!-- job definition as before -->
<job id="wordcountJob" ... />

<!-- script definition as before -->
<script id="setupScript" ... />
```

```
<job-runner id="runner" pre-action="setupScript" job="wordcountJob"/>

<task:scheduled-tasks>
  <task:scheduled ref="runner" method="call" cron="3/30 * * * * ?"/>
</task:scheduled-tasks>
```

The default value of `JobRunner`'s `run-at-startup` element is `false` so in this configuration the HDFS scripts and Hadoop jobs will not execute when the application starts. Using this configuration allows the scheduler to be the only component in the system that is responsible for executing the scripts and jobs.

This sample application can be found in the directory *hadoop/scheduling*. When running the application, we can see the (truncated) output shown in Example 11-33, where the timestamps correspond to those defined by the cron expression.

Example 11-33. Output from the scheduled wordcount application

```
removing existing input and output directories in HDFS...
copying files to HDFS...
23:20:33.664 [pool-2-thread-1] WARN  o.a.hadoop.util.NativeCodeLoader
- Unable to load native-hadoop library for your platform...
23:20:33.689 [pool-2-thread-1] WARN  org.apache.hadoop.mapred.JobClient
- No job jar file set.  User classes may not be found.
23:20:33.711 [pool-2-thread-1] INFO  o.a.h.m.lib.input.FileInputFormat
- Total input paths to process : 1
23:20:34.258 [pool-2-thread-1] INFO  org.apache.hadoop.mapred.JobClient
- Running job: job_local_0001 ...
23:20:43.978 [pool-2-thread-1] INFO  org.apache.hadoop.mapred.JobClient
-  map 100% reduce 100%
23:20:44.979 [pool-2-thread-1] INFO  org.apache.hadoop.mapred.JobClient
- Job complete: job_local_0001
23:20:44.982 [pool-2-thread-1] INFO  org.apache.hadoop.mapred.JobClient
- Counters: 22
removing existing input and output directories in HDFS...
copying files to HDFS...
23:21:03.396 [pool-2-thread-1] INFO  org.apache.hadoop.mapred.JobClient
- Running job: job_local_0001
23:21:03.397 [pool-2-thread-1] INFO  org.apache.hadoop.mapred.JobClient
- Job complete: job_local_0001
```

That's it! As you can see, the task namespace is simple to use, but it also has many features (which we will not cover) related to the thread pool policies or delegation to the CommonJ `WorkManager`. The Spring Framework reference documentation describes these features in more detail.

Scheduling MapReduce Jobs with Quartz

Quartz is a popular open source job scheduler that includes many advanced features such as clustering. In this example, we replace the Spring `TaskScheduler` used in the previous example with Quartz. The Quartz scheduler requires you to define a `Job` (aka a `JobDetail`), a `Trigger`, and a `Scheduler`, as shown in Example 11-34.

Example 11-34. Defining a Quartz scheduler to execute HDFS scripts and MapReduce jobs

```
<!-- job definition as before -->
<hdp:job id="wordcountJob" … />

<!-- script definition as before -->
<hdp:script id="setupScript" … />

<!-- simple job runner as before -->
<hdp:job-runner

<bean id="jobDetail"
      class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
  <property name="targetObject" ref="runner"/>
  <property name="targetMethod" value="run"/>
</bean>

<bean id="cronTrigger" class="org.springframework.scheduling.quartz.CronTriggerBean">
  <property name="jobDetail" ref="jobDetail"/>
  <property name="cronExpression" value="3/30 * * * * ?"/>
</bean>

<bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers" ref="cronTrigger"/>
</bean>
```

Quartz's `JobDetail` class encapsulates what code will execute when the trigger condition is satisfied. Spring's helper class `MethodInvokingJobDetailFactoryBean` will create a `JobDetail` object whose behavior is delegated to invoking the specified method on a Spring-managed object. This application can be found in the directory *hadoop/scheduling-quartz*. Running the application gives similar output to the previous Spring `TaskScheduler`-based example.

CHAPTER 12

Analyzing Data with Hadoop

While the MapReduce programming model is at the heart of Hadoop, it is low-level and as such becomes a unproductive way for developers to write complex analysis jobs. To increase developer productivity, several higher-level languages and APIs have been created that abstract away the low-level details of the MapReduce programming model. There are several choices available for writing data analysis jobs. The Hive and Pig projects are popular choices that provide SQL-like and procedural data flow-like languages, respectively. HBase is also a popular way to store and analyze data in HDFS. It is a column-oriented database, and unlike MapReduce, provides random read and write access to data with low latency. MapReduce jobs can read and write data in HBase's table format, but data processing is often done via HBase's own client API. In this chapter, we will show how to use Spring for Apache Hadoop to write Java applications that use these Hadoop technologies.

Using Hive

The previous chapter used the MapReduce API to analyze data stored in HDFS. While counting the frequency of words is relatively straightforward with the MapReduce API, more complex analysis tasks don't fit the MapReduce model as well and thus reduce developer productivity. In response to this difficulty, Facebook developed Hive as a means to interact with Hadoop in a more declarative, SQL-like manner. Hive provides a language called HiveQL to analyze data stored in HDFS, and it is easy to learn since it is similar to SQL. Under the covers, HiveQL queries are translated into multiple jobs based on the MapReduce API. Hive is now a top-level Apache project and is still heavily developed by Facebook.

While providing a deep understanding of Hive is beyond the scope of this book, the basic programming model is to create a Hive table schema that provides structure on top of the data stored in HDFS. HiveQL queries are then pared by the Hive engine, translating them into MapReduce jobs in order to execute the queries. HiveQL statements can be submitted to the Hive engine through the command line or through a component called the Hive Server, which provides access via JDBC, ODBC, or Thrift.

For more details on how to install, run, and develop with Hive and HiveQL, refer to the project website as well as the book *Programming Hive* (O'Reilly).

As with MapReduce jobs, Spring for Apache Hadoop aims to simplify Hive programming by removing the need to use command-line tools to develop and execute Hive applications. Instead, Spring for Apache Hadoop makes it easy to write Java applications that connect to a Hive server (optionally embedded), create Hive Thrift clients, and use Spring's rich JDBC support (`JdbcTemplate`) via the Hive JDBC driver.

Hello World

As an introduction to using Hive, in this section we will perform a small analysis on the Unix password file using the Hive command line. The goal of the analysis is to create a report on the number of users of a particular shell (e.g., bash or sh). To install Hive, download it from the main Hive website. After installing the Hive distribution, add its *bin* directory to your path. Now, as shown in Example 12-1, we start the Hive command-line console to execute some HiveQL commands.

Example 12-1. Analyzing a password file in the Hive command-line interface

```
$ hive
hive> drop table passwords;
hive> create table passwords (user string, passwd string, uid int, gid int,
                              userinfo string, home string, shell string)
    > ROW FORMAT DELIMITED FIELDS TERMINATED BY ':' LINES TERMINATED BY '10';
hive> load data local inpath '/etc/passwd' into table passwords;
Copying data from file:/etc/passwd
Copying file: file:/etc/passwd
Loading data to table default.passwords
OK
hive> drop table grpshell;
hive> create table grpshell (shell string, count int);
hive> INSERT OVERWRITE TABLE grpshell SELECT p.shell, count(*)
                                      FROM passwords p GROUP BY p.shell;
Total MapReduce jobs = 1
Launching Job 1 out of 1
…
Total MapReduce CPU Time Spent: 1 seconds 980 msec
hive> select * from grpshell;
OK
/bin/bash 5
/bin/false 16
/bin/sh 18
/bin/sync 1
/usr/sbin/nologin 1
Time taken: 0.122 seconds
```

You can also put the HiveQL commands in a file and execute that from the command line (Example 12-2).

Example 12-2. Executing Hive from the command line

```
$ hive -f password-analysis.hql
$ hadoop dfs -cat /user/hive/warehouse/grpshell/000000_0

/bin/bash 5
/bin/false 16
/bin/sh 18
/bin/sync 1
/usr/sbin/nologin 1
```

The Hive command line passes commands directly to the Hive engine. Hive also supports variable substitution using the notation `${hiveconf:varName}` inside the script and the command line argument `-hiveconfg varName=varValue`. To interact with Hive outside the command line, you need to connect to a Hive server using a Thrift client or over JDBC. The next section shows how you can start a Hive server on the command line or bootstrap an embedded server in your Java application.

Running a Hive Server

In a production environment, it is most common to run a Hive server as a standalone server process—potentially multiple Hive servers behind a HAProxy—to avoid some known issues with handling many concurrent client connections.[1]

If you want to run a standalone server for use in the sample application, start Hive using the command line:

```
hive --service hiveserver -hiveconf fs.default.name=hdfs://localhost:9000 \
     -hiveconf mapred.job.tracker=localhost:9001
```

Another alternative, useful for development or to avoid having to run another server, is to bootstrap the Hive server in the same Java process that will run Hive client applications. The Hadoop namespace makes embedding the Hive server a one-line configuration task, as shown in Example 12-3.

Example 12-3. Creating a Hive server with default options

```
<hive-server/>
```

By default, the hostname is `localhost` and the port is 10000. You can change those values using the `host` and `port` attributes. You can also provide additional options to the Hive server by referencing a properties file with the `properties-location` attribute or by inlining properties inside the `<hive-server/>` XML element. When the `ApplicationContext` is created, the Hive server is started automatically. If you wish to override this behavior, set the `auto-startup` element to `false`. Lastly, you can reference a specific Hadoop configuration object, allowing you to create multiple Hive servers that connect to different Hadoop clusters. These options are shown in Example 12-4.

1. *https://cwiki.apache.org/confluence/display/Hive/HiveServer2+Thrift+API*

Example 12-4. Creating and configuring a Hive server

```
<context:property-placeholder location="hadoop.properties,hive.properties" />

<configuration id="hadoopConfiguration">
  fs.default.name=${hd.fs}
  mapred.job.tracker=${mapred.job.tracker}
</configuration>

<hive-server port="${hive.port}" auto-startup="false"
             configuration-ref="hadoopConfiguration"
             properties-location="hive-server.properties">
  hive.exec.scratchdir=/tmp/hive/
</hive-server>
```

The files *hadoop.properties* and *hive.properties* are loaded from the classpath. Their combined values are shown in Example 12-5. We can use the property file *hive-server-config.properties* to configure the server; these values are the same as those you would put inside *hive-site.xml*.

Example 12-5. Properties used to configure a simple Hive application

```
hd.fs=hdfs://localhost:9000
mapred.job.tracker=localhost:9001
hive.host=localhost
hive.port=10000
hive.table=passwords
```

Using the Hive Thrift Client

The Hadoop namespace supports creating a Thrift client, as shown in Example 12-6.

Example 12-6. Creating and configuring a Hive Thrift client

```
<hive-client-factory host="${hive.host}" port="${hive.port}"/>
```

The namespace creates an instance of the class `HiveClientFactory`. Calling the method `getHiveClient` on `HiveClientFactory` will return a new instance of the `HiveClient`. This is a convenient pattern that Spring provides since the `HiveClient` is not a thread-safe class, so a new instance needs to be created inside methods that are shared across multiple threads. Some of the other parameters that we can set on the `HiveClient` through the XML namespace are the connection timeout and a collection of scripts to execute once the client connects. To use the `HiveClient`, we create a `HivePasswordRepository` class to execute the *password-analysis.hql* script used in the previous section and then execute a query against the passwords table. Adding a `<context:component-scan/>` element to the configuration for the Hive server shown earlier will automatically register the `HivePasswordRepository` class with the container by scanning the classpath for classes annotated with the Spring stereotype `@Repository` annotation. See Example 12-7.

Example 12-7. Using the Thrift HiveClient in a data access layer

```
@Repository
public class HivePasswordRepository implements PasswordRepository {

  private static final Log logger = LogFactory.getLog(HivePasswordRepository.class);

  private HiveClientFactory hiveClientFactory;
  private String tableName;

  // constructor and setters omitted

  @Override
  public Long count() {
    HiveClient hiveClient = hiveClientFactory.getHiveClient();
    try {
      hiveClient.execute("select count(*) from " + tableName);
      return Long.parseLong(hiveClient.fetchOne());
      // checked exceptions
    } catch (HiveServerException ex) {
      throw translateExcpetion(ex);
    } catch (org.apache.thrift.TException tex) {
      throw translateExcpetion(tex);
    } finally {
      try {
        hiveClient.shutdown();
      } catch (org.apache.thrift.TException tex) {
        logger.debug("Unexpected exception on shutting down HiveClient", tex);
      }
    }
  }

  @Override
  public void processPasswordFile(String inputFile) {
    // Implementation not shown
  }

  private RuntimeException translateExcpetion(Exception ex) {
    return new RuntimeException(ex);
  }
}
```

The sample code for this application is located in *./hadoop/hive*. Refer to the *readme* file in the sample's directory for more information on running the sample application. The driver for the sample application will call `HivePasswordRepository`'s `processPasswordFile` method and then its count method, returning the value `41` for our dataset. The error handling is shown in this example to highlight the data access layer development best practice of avoiding throwing checked exceptions to the calling code.

The helper class `HiveTemplate`, which provides a number of benefits that can simplify the development of using Hive programmatically. It translates the `HiveClient`'s checked exceptions and error codes into Spring's portable DAO exception hierarchy. This means that calling code does not have to be aware of Hive. The `HiveClient` is also not

thread-safe, so as with other template classes in Spring, the `HiveTemplate` provides thread-safe access to the underlying resources so you don't have to deal with the incidental complexity of the `HiveClient`'s API. You can instead focus on executing HSQL and getting results. To create a `HiveTemplate`, use the XML namespace and optionally pass in a reference to the name of the `HiveClientFactory`. Example 12-8 is a minimal configuration for the use of a new implementation of `PasswordRepository` that uses the `HiveTemplate`.

Example 12-8. Configuring a HiveTemplate

```
<context:property-placeholder location="hadoop.properties,hive.properties"/>

<configuration>
  fs.default.name=${hd.fs}
  mapred.job.tracker=${mapred.job.tracker}
</configuration>

<hive-client-factory host="${hive.host}" port="${hive.port}"/>

<hive-template/>
```

The XML namespace for `<hive-template/>` will also let you explicitly reference a `HiveClientFactory` by name using the `hive-client-factory-ref` element. Using `HiveTemplate`, the `HiveTemplatePasswordRepository` class is now much more simply implemented. See Example 12-9.

Example 12-9. PersonRepository implementation using HiveTemplate

```
@Repository
public class HiveTemplatePasswordRepository implements PasswordRepository {

  private HiveOperations hiveOperations;
  private String tableName;

  // constructor and setters omitted

  @Override
  public Long count() {
    return hiveOperations.queryForLong("select count(*) from " + tableName);
  }

  @Override
  public void processPasswordFile(String inputFile) {
    Map parameters = new HashMap();
    parameters.put("inputFile", inputFile);
    hiveOperations.query("classpath:password-analysis.hql", parameters);
  }
}
```

Note that the `HiveTemplate` class implements the `HiveOperations` interface. This is a common implementation style of Spring template classes since it facilitates unit testing, as interfaces can be easily mocked or stubbed. The helper method `queryForLong` makes

it a one liner to retrieve simple values from Hive queries. `HiveTemplate`'s query methods also let you pass a reference to a script location using Spring's `Resource` abstraction, which provides great flexibility for loading an `InputStream` from the classpath, a file, or over HTTP. The query method's second argument is used to replace substitution variables in the script with values. `HiveTemplate` also provides an execute callback method that will hand you a managed `HiveClient` instance. As with other template classes in Spring, this will let you get at the lower-level API if any of the convenience methods do not meet your needs but you will still benefit from the template's exception, translation, and resource management features.

Spring for Apache Hadoop also provides a `HiveRunner` helper class that like the `JobRunner`, lets you execute HDFS script operations before and after running a HiveQL script. You can configure the runner using the XML namespace element `<hive-runner/>`.

Using the Hive JDBC Client

The JDBC support for Hive lets you use your existing Spring knowledge of `JdbcTemplate` to interact with Hive. Hive provides a `HiveDriver` class that can be passed into Spring's `SimpleDriverDataSource`, as shown in Example 12-10.

Example 12-10. Creating and configuring a Hive JDBC-based access

```
<bean id="hiveDriver" class="org.apache.hadoop.hive.jdbc.HiveDriver" />

<bean id="dataSource" class="org.springframework.jdbc.datasource.SimpleDriverDataSource">
  <constructor-arg name="driver" ref="hiveDriver" />
  <constructor-arg name="url" value="${hive.url}"/>
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.simple.JdbcTemplate">
  <constructor-arg ref="dataSource" />
</bean>
```

`SimpleDriverDataSource` provides a simple implementation of the standard JDBC `DataSource` interface given a `java.sql.Driver` implementation. It returns a new connection for each call to the `DataSource`'s `getConnection` method. That should be sufficient for most Hive JDBC applications, since the overhead of creating the connection is low compared to the length of time for executing the Hive operation. If a connection pool is needed, it is easy to change the configuration to use Apache Commons DBCP or c3p0 connection pools.

`JdbcTemplate` brings a wide range of `ResultSet` to POJO mapping functionality as well as translating error codes into Spring's portable DAO (data access object) exception hierarchy. As of Hive 0.10, the JDBC driver supports generating meaningful error codes. This allows you to easily distinguish between catching Spring's `TransientDataAccessException` and `NonTransientDataAccessException`. *Transient* exceptions indicate that the operation can be retried and will probably succeed, whereas a *nontransient* exception indicates that retrying the operation will not succeed.

An implementation of the `PasswordRepository` using `JdbcTemplate` is shown in Example 12-11.

Example 12-11. PersonRepository implementation using JdbcTemplate

```
@Repository
public class JdbcPasswordRepository implements PasswordRepository {

  private JdbcOperations jdbcOperations;
  private String tableName;

  // constructor and setters omitted

  @Override
  public Long count() {
    return jdbcOperations.queryForLong("select count(*) from " + tableName);
  }

  @Override
  public void processPasswordFile(String inputFile) {
    // Implementation not shown.
  }
}
```

The implementation of the method processPasswordFile is somewhat lengthy due to the need to replace substitution variables in the script. Refer to the sample code for more details. Note that Spring provides the utility class `SimpleJdbcTestUtils` is part of the testing package; it's often used to execute DDL scripts for relational databases but can come in handy when you need to execute HiveQL scripts without variable substitution.

Apache Logfile Analysis Using Hive

Next, we will perform a simple analysis on Apache HTTPD logfiles using Hive. The structure of the configuration to run this analysis is similar to the one used previously to analyze the `password` file with the `HiveTemplate`. The HiveQL script shown in Example 12-12 generates a file that contains the cumulative number of hits for each URL. It also extracts the minimum and maximum hit numbers and a simple table that can be used to show the distribution of hits in a simple chart.

Example 12-12. HiveQL for basic Apache HTTPD log analysis

```
ADD JAR ${hiveconf:hiveContribJar};

DROP TABLE IF EXISTS apachelog;
CREATE TABLE apachelog(remoteHost STRING, remoteLogname STRING, user STRING, time STRING,
                       method STRING, uri STRING, proto STRING, status STRING,
                       bytes STRING, referer STRING, userAgent STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
WITH SERDEPROPERTIES (
```

```
  "input.regex" = "^([^ ]*) +([^ ]*) +([^ ]*) +\\[([^]]*)\\] +\\\"([^ ]*) ([^ ]*)
 ([^ ]*)\\\" ([^ ]*) ([^ ]*) (?:\\\"-\\\")*\\\"(.*)\\\" (.*)$",
  "output.format.string" = "%1$s %2$s %3$s %4$s %5$s %6$s %7$s %8$s %9$s %10$s %11$s")
STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH "${hiveconf:localInPath}" INTO TABLE apachelog;

-- basic filtering
-- SELECT a.uri FROM apachelog a WHERE a.method='GET' AND a.status='200';

-- determine popular URLs (for caching purposes)

INSERT OVERWRITE LOCAL DIRECTORY 'hive_uri_hits' SELECT a.uri, "\t", COUNT(*)
 FROM apachelog a GROUP BY a.uri ORDER BY uri;

-- create histogram data for charting, view book sample code for details
```

This example uses the utility library *hive-contrib.jar*, which contains a serializer/deserializer that can read and parse the file format of Apache logfiles. The *hive-contrib.jar* can be downloaded from Maven central or built directly from the source. While we have parameterized the location of the hive-contrib.jar another option is to put a copy of the jar into the Hadoop library directory on all task tracker machines. The results are placed in local directories. The sample code for this application is located in *./hadoop/hive*. Refer to the *readme* file in the sample's directory for more information on running the application. A sample of the contents of the data in the *hive_uri_hits* directory is shown in Example 12-13.

Example 12-13. The cumulative number of hits for each URI

```
/archives.html 3
/archives/000005.html 2
/archives/000021.html 1
…
/archives/000055.html 1
/archives/000064.html 2
```

The contents of the *hive_histogram* directory show that there is 1 URL that has been requested 22 times, 3 URLs were each hit 4 times, and 74 URLs have been hit only once. This gives us an indication of which URLs would benefit from being cached. The sample application shows two ways to execute the Hive script, using the `HiveTemplate` and the `HiveRunner`. The configuration for the `HiveRunner` is shown in Example 12-14.

Example 12-14. Using a HiveRunner to run the Apache Log file analysis

```
<context:property-placeholder location="hadoop.properties,hive.properties"/>

<configuration>
  fs.default.name=${hd.fs}
  mapred.job.tracker=${mapred.job.tracker}
</configuration>
```

```
<hive-server port="${hive.port}"
             properties-location="hive-server.properties"/>

<hive-client-factory host="${hive.host}" port="${hive.port}"/>

<hive-runner id="hiveRunner" run-at-startup="false" >
  <script location="apache-log-simple.hql">
    <arguments>
      hiveContribJar=${hiveContribJar}
      localInPath="./data/apache.log"
    </arguments>
  </script>
</hive-runner>
```

While the size of this dataset was very small and we could have analyzed it using Unix command-line utilities, using Hadoop lets us scale the analysis over very large sets of data. Hadoop also lets us cheaply store the raw data so that we can redo the analysis without the information loss that would normally result from keeping summaries of historical data.

Using Pig

Pig provides an alternative to writing MapReduce applications to analyze data stored in HDFS. Pig applications are written in the Pig Latin language, a high-level data processing language that is more in the spirit of using `sed` or `awk` than the SQL-like language that Hive provides. A Pig Latin script describes a sequence of steps, where each step performs a transformation on items of data in a collection. A simple sequence of steps would be to load, filter, and save data, but more complex operation—such as joining two data items based on common values—are also available. Pig can be extended by user-defined functions (UDFs) that encapsulate commonly used functionality such as algorithms or support for reading and writing well-known data formats such as Apache HTTPD logfiles. A `PigServer` is responsible for translating Pig Latin scripts into multiple jobs based on the MapReduce API and executing them.

A common way to start developing a Pig Latin script is to use the interactive console that ships with Pig, called Grunt. You can execute scripts in two different run modes. The first is the LOCAL mode, which works with data stored on the local filesystem and runs MapReduce jobs locally using an embedded version of Hadoop. The second mode, MAPREDUCE, uses HDFS and runs MapReduce jobs on the Hadoop cluster. By using the local filesystem, you can work on a small set of the data and develop your scripts iteratively. When you are satisfied with your script's functionality, you can easily switch to running the same script on the cluster over the full dataset. As an alternative to using the interactive console or running Pig from the command line, you can embed the Pig in your application. The `PigServer` class encapsulates how you can programmatically connect to Pig, execute scripts, and register functions.

Spring for Apache Hadoop makes it very easy to embed the `PigServer` in your application and to run Pig Latin scripts programmatically. Since Pig Latin does not have control flow statements such as conditional branches (if-else) or loops, Java can be useful to fill in those gaps. Using Pig programmatically also allows you to execute Pig scripts in response to event-driven activities using Spring Integration, or to take part in a larger workflow using Spring Batch.

To get familiar with Pig, we will first write a basic application to analyze the Unix password files using Pig's command-line tools. Then we show how you can use Spring for Apache Hadoop to develop Java applications that make use of Pig. For more details on how to install, run, and develop with Pig and Pig Latin, refer to the project website as well as the book *Programming Pig* (O'Reilly).

Hello World

As a Hello World exercise, we will perform a small analysis on the Unix password file. The goal of the analysis is to create a report on the number of users of a particular shell (e.g., bash or sh). Using familiar Unix utilities, you can easily see how many people are using the bash shell (Example 12-15).

Example 12-15. Using Unix utilities to count users of the bash shell

```
$ $ more /etc/passwd | grep /bin/bash
root:x:0:0:root:/root:/bin/bash
couchdb:x:105:113:CouchDB Administrator,,,:/var/lib/couchdb:/bin/bash
mpollack:x:1000:1000:Mark Pollack,,,:/home/mpollack:/bin/bash
postgres:x:116:123:PostgreSQL administrator,,,:/var/lib/postgresql:/bin/bash
testuser:x:1001:1001:testuser,,,,:/home/testuser:/bin/bash

$ more /etc/passwd | grep /bin/bash | wc -l
5
```

To perform a similar analysis using Pig, we first load the */etc/password* file into HDFS (Example 12-16).

Example 12-16. Copying /etc/password into HDFS

```
$ hadoop dfs -copyFromLocal /etc/passwd /test/passwd
$ hadoop dfs -cat /test/passwd

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
…
```

To install Pig, download it from the main Pig website. After installing the distribution, you should add the Pig distribution's *bin* directory to your path and also set the environment variable `PIG_CLASSPATH` to point to the Hadoop configuration directory (e.g., `export PIG_CLASSPATH=$HADOOP_INSTALL/conf/`).

Now we start the Pig interactive console, Grunt, in LOCAL mode and execute some Pig Latin commands (Example 12-17).

Example 12-17. Executing Pig Latin commands using Grunt

```
$ pig -x local
grunt> passwd = LOAD '/test/passwd' USING PigStorage(':') \
       AS (username:chararray, password:chararray, uid:int, gid:int, userinfo:chararray,
           home_dir:chararray, shell:chararray);
grunt> grouped_by_shell = GROUP passwd BY shell;
grunt> password_count = FOREACH grouped_by_shell GENERATE group, COUNT(passwd);
grunt> STORE password_count into '/tmp/passwordAnalysis';
grunt> quit
```

Since the example dataset is small, all the results fit in a single tab-delimited file, as shown in Example 12-18.

Example 12-18. Command execution result

```
$ hadoop dfs -cat /tmp/passwordAnalysis/part-r-00000

/bin/sh 18
/bin/bash 5
/bin/sync 1
/bin/false 16
/usr/sbin/nologin 1
```

The general flow of the data transformations taking place in the Pig Latin script is as follows. The first line loads the data from the HDFS file, */test/passwd*, into the variable named `passwd`. The `LOAD` command takes the location of the file in HDFS, as well as the format by which the lines in the file should be broken up, in order to create a dataset (aka, a Pig relation). In this example, we are using the `PigStorage` function to load the text file and separate the fields based on a colon character. Pig can apply a schema to the columns that are in the file by defining a name and a data type to each column. With the input dataset assigned to the variable `passwd`, we can now perform operations on the dataset to transform it into other derived datasets. We create the dataset `grouped_by_shell` using the `GROUP` operation. The `grouped_by_shell` dataset will have the shell name as the key and a collection of all records in the `passwd` dataset that have that shell value. If we had used the `DUMP` operation to view the contents of the `grouped_by_shell` dataset for the `/bin/bash` key, we would see the result shown in Example 12-19.

Example 12-19. Group-by-shell dataset

```
(/bin/bash,{(testuser,x,1001,1001,testuser,,,,,/home/testuser,/bin/bash),
            (root,x,0,0,root,/root,/bin/bash),
            (couchdb,x,105,113,CouchDB Administrator,,,,/var/lib/couchdb,/bin/bash),
            (mpollack,x,1000,1000,Mark Pollack,,,,/home/mpollack,/bin/bash),
            (postgres,x,116,123,PostgreSQL administrator,,,,/var/lib/postgresql,/bin/bash)
           })
```

The key is the shell name and the value is a collection, or bag, of password records that have the same key. In the next line, the expression `FOREACH grouped_by_shell GENERATE` will apply an operation on each record of the `grouped_by_shell` dataset and generate a new record. The new dataset that is created will group together all the records with the same key and count them.

We can also parameterize the script to avoid hardcoding values—for example, the input and output locations. In Example 12-20, we put all the commands entered into the interactive console into a file named *password-analysis.pig*, parameterized by the variables `inputFile` and `outputDir`.

Example 12-20. Parameterized Pig script

```
passwd = LOAD '$inputFile' USING PigStorage(':')
         AS (username:chararray, password:chararray, uid:int, gid:int, userinfo:chararray,
             home_dir:chararray, shell:chararray);
grouped_by_shell = GROUP passwd BY shell;
password_count = FOREACH grouped_by_shell GENERATE group, COUNT(passwd);
STORE password_count into '$outputDir';
```

To run this script in the interactive console, use the `run` command, as shown in Example 12-21.

Example 12-21. Running a parameterized Pig script in Grunt

```
grunt> run -param inputFile=/test/passwd outputDir=/tmp/passwordAnalysis
  password-analysis.pig
grunt> exit

$ hadoop dfs -cat /tmp/passwordAnalysis/part-r-00000

/bin/sh 18
/bin/bash 5
/bin/sync 1
/bin/false 16
/usr/sbin/nologin 1
```

Or, to run the script directly in the command line, see Example 12-22.

Example 12-22. Running a parameterized Pig script on the command line

```
$ pig -file password-analysis.pig -param inputFile=/test/passwd
  -param outputDir=/tmp/passwordAnalysis
```

Running a PigServer

Now we'll shift to using a more structured and programmatic way of running Pig scripts. Spring for Apache Hadoop makes it easy to declaratively configure and create a `PigServer` in a Java application, much like the Grunt shell does under the covers. Running a `PigServer` inside the Spring container will let you parameterize and externalize properties that control what Hadoop cluster to execute your scripts against,

properties of the `PigServer`, and arguments passed into the script. Spring's XML namespace for Hadoop makes it very easy to create and configure a `PigServer`. As Example 12-23 demonstrates, this configuration is done in the same way as the rest of your application configuration. The location of the optional Pig initialization script can be any Spring resource URL, located on the filesystem, classpath, HDFS, or HTTP.

Example 12-23. Configuring a PigServer

```
<context:property-placeholder location="hadoop.properties" />

<configuration>
  fs.default.name=${hd.fs}
  mapred.job.tracker=${mapred.job.tracker}
</configuration>

<pig-factory properties-location="pig-server.properties"
  <script location="initialization.pig">
    <arguments>
      inputFile=${initInputFile}
    </arguments>
  </script>
<pig-factory/>
```

In Example 12-24, the script is located from the classpath and the variables to be replaced are contained in the property file *hadoop.properties*.

Example 12-24. hadoop.properties

```
hd.fs=hdfs://localhost:9000
mapred.job.tracker=localhost:9001
inputFile=/test/passwd
outputDir=/tmp/passwordAnalysis
```

Some of the other attributes on the `<pig-factory/>` namespace are `properties-location`, which references a properties file to configure properties of the `PigServer`; `job-tracker`, which sets the location of the job tracker used to a value different than that used in the Hadoop configuration; and `job-name`, which sets the root name of the MapReduce jobs created by Pig so they can be easily identified as belonging to this script.

Since the `PigServer` class is not a thread-safe object and there is state created after each execution that needs to be cleaned up, the `<pig-factory/>` namespace creates an instance of a `PigServerFactory` so that you can easily create new `PigServer` instances as needed. Similar in purpose to `JobRunner` and `HiveRunner`, the `PigRunner` helper class to provide a convenient way to repeatedly execute Pig jobs and also execute HDFS scripts before and after their execution. The configuration of the `PigRunner` is shown in Example 12-25.

Example 12-25. Configuring a PigRunner

```
<pig-runner id="pigRunner"
            pre-action="hdfsScript"
            run-at-startup="true" >
```

```
  <script location="password-analysis.pig">
    <arguments>
      inputDir=${inputDir}
      outputDir=${outputDir}
    </arguments>
  </script>
</pig-runner>
```

We set the `run-at-startup` element to `true`, enabling the Pig script to be executed when the Spring `ApplicationContext` is started (the default is `false`). The sample application is located in the directory *hadoop/pig*. To run the sample password analysis application, run the commands shown in Example 12-26.

Example 12-26. Command to build and run the Pig scripting example

```
$ cd hadoop/pig
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/pigApp
```

Since the `PigServerFactory` and `PigRunner` classes are Spring-managed objects, they can also be injected into any other object managed by Spring. It is often convenient to inject the `PigRunner` helper class to ensure that a new instance of the `PigServer` is created for each execution of the script and that its resources used are cleaned up after execution. For example, to run a Pig job asynchronously as part of a service layer in an application, we inject the `PigRunner` and use Spring's `@Async` annotation (Example 12-27).

Example 12-27. Dependency injection of a PigRunner to execute a Pig job asynchronously

```
@Component
public class AnalysisService {

  private PigRunner pigRunner;

  @Autowired
  public AnalysisService(PigRunner pigRunner)
    this.pigRunner = pigRunner;
  }

  @Async
  public void performAnalysis() {
    pigRunner.call();
  }
}
```

Controlling Runtime Script Execution

To have more runtime control over what Pig scripts are executed and the arguments passed into them, we can use the `PigTemplate` class. As with other template classes in Spring, `PigTemplate` manages the underlying resources on your behalf, is thread-safe once configured, and will translate Pig errors and exceptions into Spring's portable DAO exception hierarchy. Spring's DAO exception hierarchy makes it easier to work

across different data access technologies without having to catch exceptions or look for return codes, which are specific to each technology. Spring's DAO exception hierarchy also helps to separate out nontransient and transient exceptions. In the case of a transient exception being thrown, the failed operation might be able to succeed if it is retried again. Using retry advice on a data access layer via Spring's AOP (aspect-oriented programming) support is one way to implement this functionality. Since Spring's JDBC helper classes also perform the same exception translation, exceptions thrown in any Hive-based data access that uses Spring's JDBC support will also map into the DAO hierarchy. While switching between Hive and Pig is not a trivial task, since analysis scripts need to be rewritten, you can at least insulate calling code from the differences in implementation between a Hive-based and a Pig-based data access layer. It will also allow you to more easily mix calls to Hive- and Pig-based data access classes and handle errors in a consistent way.

To configure a `PigTemplate`, create a `PigServerFactory` definition as before and add a `<pig-template/>` element (Example 12-28). We can define common configuration properties of the `PigServer` in a properties file specified by the `properties-location` element. Then reference the template inside a DAO or repository class—in this case, `PigPasswordRepository`.

Example 12-28. Configuring a PigTemplate

```
<pig-factory properties-location="pig-server.properties"/>

<pig-template/>

<beans:bean id="passwordRepository"
            class="com.oreilly.springdata.hadoop.pig.PigPasswordRepository">
  <beans:constructor-arg ref="pigTemplate"/>
</beans:bean>
```

The `PigPasswordRepository` (Example 12-29) lets you pass in the input file at runtime. The method `processPasswordFiles` shows you one way to programmatically process multiple files in Java. For example, you may want to select a group of input files based on a complex set of criteria that cannot be specified inside Pig Latin or a Pig user-defined function. Note that the `PigTemplate` class implements the `PigOperations` interface. This is a common implementation style of Spring template classes since it facilitates unit testing, as interfaces can be easily mocked or stubbed.

Example 12-29. Pig-based PasswordRepository

```
public class PigPasswordRepository implements PasswordRepository {

  private PigOperations pigOperations;
  private String pigScript = "classpath:password-analysis.pig";

  // constructor and setters omitted

  @Override
  public void processPasswordFile(String inputFile) {
```

```
    Assert.notNull(inputFile);
    String outputDir =
      PathUtils.format("/data/password-repo/output/%1$tY/%1$tm/%1$td/%1$tH/%1$tM/%1$tS");
    Properties scriptParameters = new Properties();
    scriptParameters.put("inputDir", inputFile);
    scriptParameters.put("outputDir", outputDir);
    pigOperations.executeScript(pigScript, scriptParameters);
  }

  @Override
  public void processPasswordFiles(Collection<String> inputFiles) {
    for (String inputFile : inputFiles) {
      processPasswordFile(inputFile);
    }
  }
}
```

The pig script *password-analysis.pig* is loaded via Spring's resource abstraction, which in this case is loaded from the classpath. To run an application that uses the `PigPasswordRepository`, use the commands in Example 12-30.

Example 12-30. Building and running the Pig scripting example that uses PigPasswordRepository

```
$ cd hadoop/pig
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/pigAppWithRepository
```

The essential pieces of code that are executed by this application are shown in Example 12-31.

Example 12-31. Using the PigPasswordRepository

```
PasswordRepository repo = context.getBean(PigPasswordRepository.class);
repo.processPasswordFile("/data/passwd/input")

Collection<String> files = new ArrayList<String>();
files.add("/data/passwd/input");
files.add("/data/passwd/input2");
repo.processPasswordFiles(files);
```

Calling Pig Scripts Inside Spring Integration Data Pipelines

To run a Pig Latin script inside of a Spring Integration data pipeline, we can reference the `PigPasswordRepository` in a Spring Integration service activator definition (Example 12-32).

Example 12-32. Invoking a Pig script within a Spring Integration data pipeline

```
<bean id="passwordService" class="com.oreilly.springdata.hadoop.pig.PasswordService">
  <constructor-arg ref="passwordRepository" />
</bean>

<int:service-activator input-channel="exampleChannel" ref="passwordService" />
```

Whether the service activator is executed asynchronously or synchronously depends on the type of input channel used. If it is a `DirectChannel` (the default), then it will be executed synchronously; if it is an `ExecutorChannel`, it will be executed asynchronously, delegating the execution to a `TaskExecutor`. The service activator class, `PasswordService`, is shown in Example 12-33.

Example 12-33. Spring Integration service activator to execute a Pig analysis job

```
public class PasswordService {

  private PasswordRepository passwordRepository;

  // constructor omitted

  @ServiceActivator
  public void process(@Header("hdfs_path") String inputDir) {
    passwordRepository.processPasswordFile(inputDir);
  }
}
```

The process method's argument will be taken from the header of the Spring Integration message. The value of the method argument is the value associated with the key `hdfs_path` in the message header. This header value is populated by a `MessageHandler` implementation, such as the `FsShellWritingMessageHandler` used previously, and needs to be called before the service activator in the data processing pipeline.

The examples in this section show that there is a steady progression from creating a simple Pig-based application that runs a script, to executing scripts with runtime parameter substitution, to executing scripts within a Spring Integration data pipeline. The section "Hadoop Workflows" on page 238 will show you how to orchestrate the execution of a Pig script inside a larger collection of steps using Spring Batch.

Apache Logfile Analysis Using Pig

Next, we will perform a simple analysis on an Apache HTTPD logfile and show the use of a custom loader for Apache logfiles. As you can see in Example 12-34, the structure of the configuration to run this analysis is similar to the one used previously to analyze the `password` file.

Example 12-34. Configuration to analyze an Apache HTTD logfile using Pig

```
<context:property-placeholder location="hadoop.properties,pig-analysis.properties"/>

<configuration>
  fs.default.name=${hd.fs}
  mapred.job.tracker=${mapred.job.tracker}
</configuration>

<pig-factory/>

<script id="hdfsScript" location="copy-files.groovy">
```

```
  <property name="localSourceFile" value="${pig.localSourceFile}"/>
  <property name="inputDir" value="${pig.inputPath}"/>
  <property name="outputDir" value="${pig.outputPath}"/>
</script>
```

The *copy-files.groovy* script is responsible for copying a sample logfile into HDFS, removing the content of the output path.

The Pig script generates a file that contains the cumulative number of hits for each URL and is intended to be a starting point for a more comprehensive analysis. The script also extracts minimum and maximum hit numbers and a simple table that can be used to show the distribution of hits in a simple chart. Pig makes it easy to filter and preprocess the data on a variety of criteria—for example, retain only GET requests that were successful and remove GET requests for images. The Pig script and the `PigRunner` configuration that will run the script when the Spring `ApplicationContext` starts are shown in Example 12-35 and Example 12-36, respectively.

Example 12-35. Pig script for basic Apache HTTPD log analysis

```
REGISTER $piggybanklib;
DEFINE LogLoader org.apache.pig.piggybank.storage.apachelog.CombinedLogLoader();
logs = LOAD '$inputPath' USING LogLoader AS (remoteHost, remoteLogname, user, time, \
       method, uri, proto, status, bytes, referer, userAgent);

-- determine popular URLs (for caching purposes for example)
byUri = ORDER logs BY uri;
byUri = GROUP logs BY uri;

uriHits = FOREACH byUri GENERATE group AS uri, COUNT(logs.uri) AS numHits;
STORE uriHits into '$outputPath/pig_uri_hits';

-- create histogram data for charting, view book sample code for details
```

Example 12-36. PigRunner configuration to analyze Apache HTTPD files

```
<pig-runner id="pigRunner"
            pre-action="hdfsScript"
            run-at-startup="true" >
  <script location="apache-log-simple.pig">
    <arguments>
      piggybanklib=${pig.piggybanklib}
      inputPath=${pig.inputPath}
      outputPath=${pig.outputPath}
    </arguments>
  </script>
</pig-runner>
```

The arguments to the Pig script specify the location of a jar file that contains the custom loader to read and parse Apache logfiles and the location of the input and output paths. To parse the Apache HTTPD logfiles, we will use a custom loader provided as part of the Pig distribution. It is distributed as source code as part of the Piggybank project. The compiled Piggybank jar file is provided in the sample application's *lib* directory.

Using HBase

HBase is a distributed column-oriented database. It models data as tables, which are then stored in HDFS and can scale to support tables with billions of rows and millions of columns. Like Hadoop's HDFS and MapReduce, HBase is modeled on technology developed at Google. In the case of HBase, it is Google's BigTable technology, which was described in a research paper in 2006. Unlike MapReduce, HBase provides near real-time key-based access to data, and therefore can be used in interactive, non-batch-based applications.

The HBase data model consists of a table identified by a unique key with associated columns. These columns are grouped together into column families so that data that is often accessed together can be stored together on disk to increase I/O performance. The data stored in a column is a collection of key/value pairs, not a single value as is commonly the case in a relational database. A schema is used to describe the column families and needs to be defined in advance, but the collection of key/value pairs stored as values does not. This gives the system a great amount of flexibility to evolve. There are many more details to the data model, which you must thoroughly understand in order to use HBase effectively. The book *HBase: The Definitive Guide* (O'Reilly) is an excellent reference to HBase and goes into great detail about the data model, architecture, API, and administration of HBase.

Spring for Apache Hadoop provides some basic, but quite handy, support for developing HBase applications, allowing you to easily configure your connection to HBase and provide thread-safe data access to HBase tables, as well as a lightweight object-to-column data mapping functionality.

Hello World

To install HBase, download it from the main Hive website. After installing the distribution, you start the HBase server by executing the *start-hbase.sh* script in the *bin* directory. As with Pig and Hive, HBase comes with an interactive console, which you can start by executing the command `hbase shell` in the *bin* directory.

Once inside the interactive console, you can start to create tables, define column families, and add rows of data to a specific column family. Example 12-37 demonstrates creating a user table with two column families, inserting some sample data, retrieving data by key, and deleting a row.

Example 12-37. Using the HBase interactive console

```
$ ./bin/hbase shell
> create 'users', { NAME => 'cfInfo'}, { NAME => 'cfStatus' }
> put 'users', 'row-1', 'cfInfo:qUser', 'user1'
> put 'users', 'row-1', 'cfInfo:qEmail', 'user1@yahoo.com'
> put 'users', 'row-1', 'cfInfo:qPassword', 'user1pwd'
> put 'users', 'row-1', 'cfStatus:qEmailValidated', 'true'
```

```
> scan 'users'
ROW                   COLUMN+CELL
 row-1                column=cfInfo:qEmail, timestamp=1346326115599, value=user1
                      @yahoo.com
 row-1                column=cfInfo:qPassword, timestamp=1346326128125, value=us
                      er1pwd
 row-1                column=cfInfo:qUser, timestamp=1346326078830, value=user1
 row-1                column=cfStatus:qEmailValidated, timestamp=1346326146784,
                      value=true
1 row(s) in 0.0520 seconds
> get 'users', 'row-1'
COLUMN                CELL
 cfInfo:qEmail        timestamp=1346326115599, value=user1@yahoo.com
 cfInfo:qPassword     timestamp=1346326128125, value=user1pwd
 cfInfo:qUser         timestamp=1346326078830, value=user1
 cfStatus:qEmailValid timestamp=1346326146784, value=true
 ated
4 row(s) in 0.0120 seconds

> deleteall 'users', 'row-1'
```

The two column families created are named `cfInfo` and `cfStatus`. The key names, called *qualifiers* in HBase, that are stored in the `cfInfo` column family are the username, email, and password. The `cfStatus` column family stores other information that we do not frequently need to access, along with the data stored in the `cfInfo` column family. As an example, we place the status of the email address validation process in the `cfStatus` column family, but other data—such as whether the user has participated in any online surveys—is also a candidate for inclusion. The `deleteall` command deletes all data for the specified table and row.

Using the HBase Java Client

There are many client API options to interact with HBase. The Java client is what we will use in this section but REST, Thrift, and Avro clients are also available. The `HTable` class is the main way in Java to interact with HBase. It allows you to put data into a table using a `Put` class, get data by key using a `Get` class, and delete data using a `Delete` class. You query that data using a `Scan` class, which lets you specify key ranges as well as filter criteria. Example 12-38 puts a row of user data under the key `user1` into the user table from the previous section.

Example 12-38. HBase Put API

```
Configuration configuration = new Configuration(); // Hadoop configuration object
HTable table = new HTable(configuration, "users");

Put p = new Put(Bytes.toBytes("user1"));
p.add(Bytes.toBytes("cfInfo"), Bytes.toBytes("qUser"), Bytes.toBytes("user1"));
p.add(Bytes.toBytes("cfInfo"), Bytes.toBytes("qEmail"), Bytes.toBytes("user1@yahoo.com"));
p.add(Bytes.toBytes("cfInfo"), Bytes.toBytes("qPassword"), Bytes.toBytes("user1pwd"));
table.put(p);
```

The HBase API requires that you work with the data as byte arrays and not other primitive types. The `HTable` class is also not thread safe, and requires you to carefully manage the underlying resources it uses and catch HBase-specific exceptions. Spring's `HBaseTemplate` class provides a higher-level abstraction for interacting with HBase. As with other Spring template classes, it is thread-safe once created and provides exception translation into Spring's portable data access exception hierarchy. Similar to `JdbcTemplate`, it provides several callback interfaces, such as `TableCallback`, `RowMapper`, and `ResultsExtractor`, that let you encapsulate commonly used functionality, such as mapping HBase result objects to POJOs.

The `TableCallback` callback interface provides the foundation for the functionality of `HBaseTemplate`. It performs the table lookup, applies configuration settings (such as when to flush data), closes the table, and translates any thrown exceptions into Spring's DAO exception hierarchy. The `RowMapper` callback interface is used to map one row from the HBase query `ResultScanner` into a POJO. `HBaseTemplate` has several overloaded `find` methods that take additional criteria and that automatically loop over HBase's `ResultScanner` "result set" object, converting each row to a POJO, and return a list of mapped objects. See the Javadoc API for more details. For more control over the mapping process—for example, when one row does not directly map onto one POJO—the `ResultsExtractor` interface hands you the `ResultScanner` object so you can perform the iterative result set processing yourself.

To create and configure the `HBaseTemplate`, create a `HBaseConfiguration` object and pass it to `HBaseTemplate`. Configuring a `HBaseTemplate` using Spring's Hadoop XML namespace is demonstrated in Example 12-39, but it is also easy to achieve programmatically in pure Java code.

Example 12-39. Configuring an HBaseTemplate

```
<configuration>
  fs.default.name=hdfs://localhost:9000
</configuration>

<hbase-configuration configuration-ref="hadoopConfiguration" />

<beans:bean id="hbaseTemplate" class="org.springframework.data.hadoop.hbase.HbaseTemplate">
  <beans:property name="configuration" ref="hbaseConfiguration" />
</beans:bean>
```

A `HBaseTemplate`-based `UserRepository` class that finds all users and also adds users to HBase is shown in Example 12-40.

Example 12-40. HBaseTemplate-based UserRepository class

```
@Repository
public class UserRepository {

  public static final byte[] CF_INFO = Bytes.toBytes("cfInfo");

  private HbaseTemplate hbaseTemplate;
```

```
  private String tableName = "users";
  private byte[] qUser = Bytes.toBytes("user");
  private byte[] qEmail = Bytes.toBytes("email");
  private byte[] qPassword = Bytes.toBytes("password");

  // constructor omitted

  public List<User> findAll() {
    return hbaseTemplate.find(tableName, "cfInfo", new RowMapper<User>() {
      @Override
      public User mapRow(Result result, int rowNum) throws Exception {
        return new User(Bytes.toString(result.getValue(CF_INFO, qUser)),
                        Bytes.toString(result.getValue(CF_INFO, qEmail)),
                        Bytes.toString(result.getValue(CF_INFO, qPassword)));
      }
    });
  }

  public User save(final String userName, final String email, final String password) {
    return hbaseTemplate.execute(tableName, new TableCallback<User>() {
      public User doInTable(HTable table) throws Throwable {
        User user = new User(userName, email, password);
        Put p = new Put(Bytes.toBytes(user.getName()));
        p.add(CF_INFO, qUser, Bytes.toBytes(user.getName()));
        p.add(CF_INFO, qEmail, Bytes.toBytes(user.getEmail()));
        p.add(CF_INFO, qPassword, Bytes.toBytes(user.getPassword()));
        table.put(p);
        return user;
      }
    });
  }
}
```

In this example, we used an anonymous inner class to implement the `TableCallback` and `RowMapper` interfaces, but creating standalone classes is a common implementation strategy that lets you reuse mapping logic across various parts of your application. While you can develop far more functionality with HBase to make it as feature-rich as Spring's MongoDB support, we've seen that the basic plumbing for interacting with HBase available with Spring Hadoop at the time of this writing simplifies HBase application development. In addition, HBase allows for a consistent configuration and programming model that you can further use and extend across Spring Data and other Spring-related projects.

CHAPTER 13

Creating Big Data Pipelines with Spring Batch and Spring Integration

The goal of Spring for Apache Hadoop is to simplify the development of Hadoop applications. Hadoop applications involve much more than just executing a single MapReduce job and moving a few files into and out of HDFS as in the wordcount example. There is a wide range of functionality needed to create a real-world Hadoop application. This includes collecting event-driven data, writing data analysis jobs using programming languages such as Pig, scheduling, chaining together multiple analysis jobs, and moving large amounts of data between HDFS and other systems such as databases and traditional filesystems.

Spring Integration provides the foundation to coordinate event-driven activities—for example, the shipping of logfiles, processing of event streams, real-time analysis, or triggering the execution of batch data analysis jobs. Spring Batch provides the framework to coordinate coarse-grained steps in a workflow, both Hadoop-based steps and those outside of Hadoop. Spring Batch also provides efficient data processing capabilities to move data into and out of HDFS from diverse sources such as flat files, relational databases, or NoSQL databases.

Spring for Apache Hadoop in conjunction with Spring Integration and Spring Batch provides a comprehensive and consistent programming model that can be used to implement Hadoop applications that span this wide range of functionality. Another product, Splunk, also requires a wide range of functionality to create real-world big data pipeline solutions. Spring's support for Splunk helps you to create complex Splunk applications and opens the door for solutions that mix these two technologies.

Collecting and Loading Data into HDFS

The examples demonstrated until now have relied on a fixed set of data files existing in a local directory that get copied into HDFS. In practice, files that are to be loaded into HDFS are continuously generated by another process, such as a web server. The

contents of a local directory get filled up with rolled-over logfiles, usually following a naming convention such as *myapp-timestamp.log*. It is also common that logfiles are being continuously created on remote machines, such as a web farm, and need to be transferred to a separate machine and loaded into HDFS. We can implement these use cases by using Spring Integration in combination with Spring for Apache Hadoop.

In this section, we will provide a brief introduction to Spring Integration and then implement an application for each of the use cases just described. In addition, we will show how Spring Integration can be used to process and load into HDFS data that comes from an event stream. Lastly, we will show the features available in Spring Integration that enable rich runtime management of these applications through JMX (Java management extensions) and over HTTP.

An Introduction to Spring Integration

Spring Integration is an open source Apache 2.0 licensed project, started in 2007, that supports writing applications based on established enterprise integration patterns. These patterns provide the key building blocks to develop integration applications that tie new and existing system together. The patterns are based upon a messaging model in which messages are exchanged within an application as well as between external systems. Adopting a messaging model brings many benefits, such as the logical decoupling between components as well as physical decoupling; the consumer of messages does not need to be directly aware of the producer. This decoupling makes it easier to build integration applications, as they can be developed by assembling individual building blocks together. The messaging model also makes it easier to test the application, since individual blocks can be tested first in isolation from other components. This allows bugs to be found earlier in the development process rather than later during distributed system testing, where tracking down the root cause of a failure can be very difficult. The key building blocks of a Spring Integration application and how they relate to each other is shown in Figure 13-1.

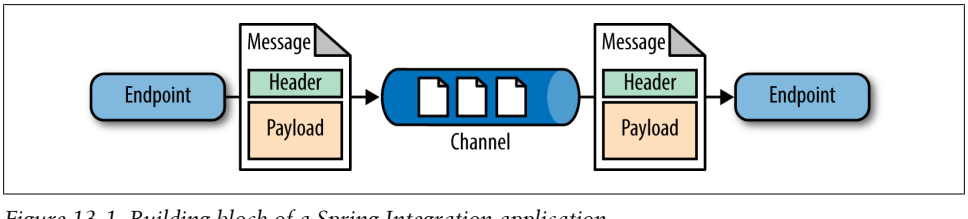


Figure 13-1. Building block of a Spring Integration application

Endpoints are producers or consumers of messages that are connected through channels. Messages are a simple data structure that contains key/value pairs in a header and an arbitrary object type for the payload. Endpoints can be adapters that communicate with external systems such as email, FTP, TCP, JMS, RabbitMQ, or syslog, but can also be operations that act on a message as it moves from one channel to another.

Common messaging operations that are supported in Spring Integration are routing to one or more channels based on the headers of message, transforming the payload from a string to a rich data type, and filtering messages so that only those that pass the filter criteria are passed along to a downstream channel. Figure 13-2 is an example taken from a joint Spring/C24 project in the financial services industry that shows the type of data processing pipelines that can be created with Spring Integration.

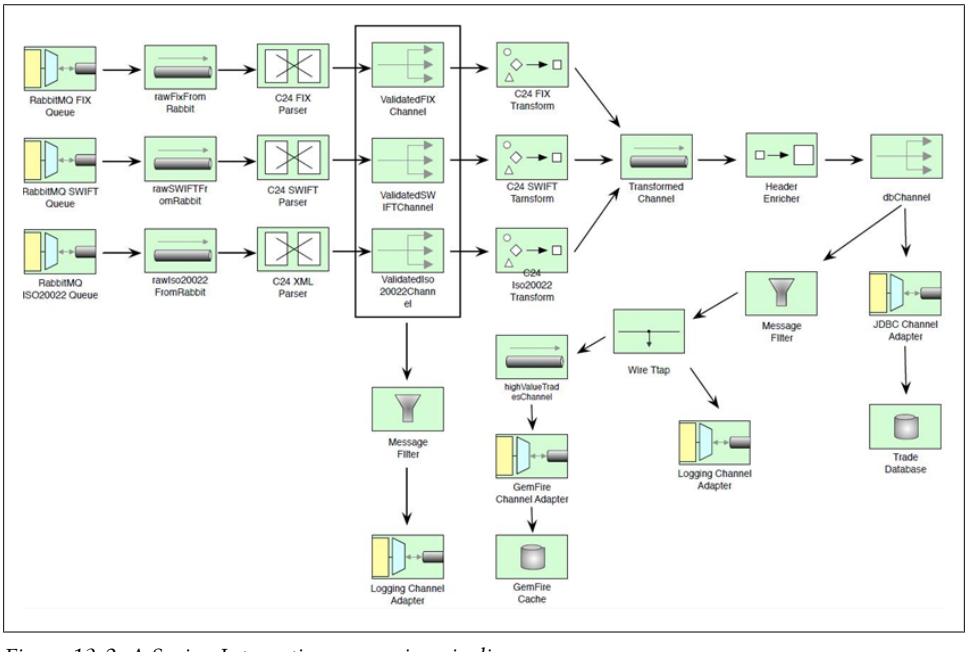


Figure 13-2. A Spring Integration processing pipeline

This diagram shows financial trade messages being received on the left via three RabbitMQ adapters that correspond to three external sources of trade data. The messages are then parsed, validated, and transformed into a canonical data format. Note that this format is not required to be XML and is often a POJO. The message header is then enriched, and the trade is stored into a relational database and also passed into a filter. The filter selects only high-value trades that are subsequently placed into a GemFire-based data grid where real-time processing can occur. We can define this processing pipeline declaratively using XML or Scala, but while most of the application can be declaratively configured, any components that you may need to write are POJOs that can be easily unit-tested.

In addition to endpoints, channels, and messages, another key component of Spring Integration is its management functionality. You can easily expose all components in a data pipeline via JMX, where you can perform operations such as stopping and starting adapters. The control bus component allows you to send in small fragments of code—for example, using Groovy—that can take complex actions to modify the state

of the system, such as changing filter criteria or starting and stopping adapters. The control bus is then connected to a middleware adapter so it can receive code to execute; HTTP and message-oriented middleware adapters are common choices.

We will not be able to dive into the inner workings of Spring Integration in great depth, nor cover every feature of the adapters that are used, but you should end up with a good feel for how you can use Spring Integration in conjunction with Spring for Apache Hadoop to create very rich data pipeline solutions. The example applications developed here contain some custom code for working with HDFS that is planned to be incorporated into the Spring Integration project. For additional information on Spring Integration, consult the project website, which contains links to extensive reference documentation, sample applications, and links to several books on Spring Integration.

Copying Logfiles

Copying logfiles into Hadoop as they are continuously generated is a common task. We will create two applications that continuously load generated logfiles into HDFS. One application will use an inbound file adapter to poll a directory for files, and the other will poll an FTP site. The outbound adapter writes to HDFS, and its implementation uses the `FsShell` class provided by Spring for Apache Hadoop, which was described in "Scripting HDFS on the JVM" on page 187. The diagram for this data pipeline is shown in Figure 13-3.

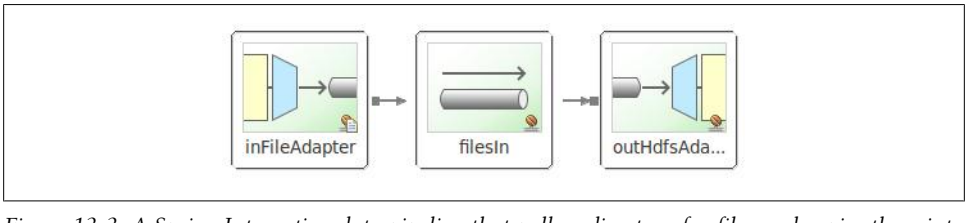


Figure 13-3. A Spring Integration data pipeline that polls a directory for files and copies them into HDFS

The file inbound adapter is configured with the directory to poll for files as well as the filename pattern that determines what files will be detected by the adapter. These values are externalized into a properties file so they can easily be changed across different runtime environments. The adapter uses a poller to check the directory since the filesystem is not an event-driven source. There are several ways you can configure the poller, but the most common are to use a fixed delay, a fixed rate, or a cron expression. In this example, we do not make use of any additional operations in the pipeline that would sit between the two adapters, but we could easily add that functionality if required. The configuration file to configure this data pipeline is shown in Example 13-1.

Example 13-1. Defining a data pipeline that polls for files in a directory and loads them into HDFS

```
<context:property-placeholder location="hadoop.properties,polling.properties"/>

<hdp:configuration id="hadoopConfiguration">fs.default.name=${hd.fs}</hdp:configuration>

<int:channel id="filesIn"/>

<file:inbound-channel-adapter id="inFileAdapter"
                              channel="filesIn"
                              directory="${polling.directory}"
                              filename-pattern="${polling.fileNamePattern}">
  <int:poller id="poller" fixed-delay="${polling.fixedDelay}"/>
</file:inbound-channel-adapter>


<int:outbound-channel-adapter id="outHdfsAdapter"
                              channel="filesIn"
                              ref="fsShellWritingMessagingHandler" >

<bean id="fsShellWritingMessagingHandler"
      class="com.oreilly.springdata.hadoop.filepolling.FsShellWritingMessageHandler">
  <constructor-arg value="${polling.destinationHdfsDirectory}"/>
  <constructor-arg ref="hadoopConfiguration"/>
</bean>
```

The relevant configuration parameters for the pipeline are externalized in the `polling.properties` file, as shown in Example 13-2.

Example 13-2. The externalized properties for polling a directory and loading them into HDFS

```
polling.directory=/opt/application/logs
polling.fixedDelay=5000
polling.fileNamePattern=*.txt
polling.destinationHdfsDirectory=/data/application/logs
```

This configuration will poll the directory */opt/application/logs* every five seconds and look for files that match the pattern **.txt*. By default, duplicate files are prevented when we specify a `filename-pattern`; the state is kept in memory. A future enhancement of the file adapter is to persistently store this application state. The `FsShellWritingMessageHandler` class is responsible for copying the file into HDFS using `FsShell`'s `copyFromLocal` method. If you want to remove the files from the polling directory after the transfer, then you set the property `deleteSourceFiles` on `FsShellWritingMessageHandler` to `true`. You can also lock files to prevent them from being picked up concurrently if more than one process is reading from the same directory. See the Spring Integration reference guide for more information.

To build and run this application, use the commands shown in Example 13-3.

Example 13-3. Command to build and run the file polling example

```
$ cd hadoop/file-polling
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/filepolling
```

The relevant parts of the output are shown in Example 13-4.

Example 13-4. Output from running the file polling example

```
03:48:44.187 [main] INFO
  c.o.s.hadoop.filepolling.FilePolling - File Polling Application Running
03:48:44.191 [task-scheduler-1] DEBUG o.s.i.file.FileReadingMessageSource - \
Added to queue: [/opt/application/logs/file_1.txt]
03:48:44.215 [task-scheduler-1] INFO  o.s.i.file.FileReadingMessageSource - \
Created message: [[Payload=/opt/application/logs/file_1.txt]
03:48:44.215 [task-scheduler-1] DEBUG o.s.i.e.SourcePollingChannelAdapter - \
Poll resulted in Message: [Payload=/opt/application/logs/file_1.txt]
03:48:44.215 [task-scheduler-1] DEBUG o.s.i.channel.DirectChannel - \
preSend on channel 'filesIn', message: [Payload=/opt/application/logs/file_1.txt]
03:48:44.310 [task-scheduler-1] INFO  c.o.s.h.f.FsShellWritingMessageHandler - \
sourceFile = /opt/application/logs/file_1.txt
03:48:44.310 [task-scheduler-1] INFO  c.o.s.h.f.FsShellWritingMessageHandler - \
resultFile = /data/application/logs/file_1.txt
03:48:44.462 [task-scheduler-1] DEBUG o.s.i.channel.DirectChannel - \
postSend (sent=true) on channel 'filesIn', \
message: [Payload=/opt/application/logs/file_1.txt]
03:48:49.465 [task-scheduler-2] DEBUG o.s.i.e.SourcePollingChannelAdapter - \
Poll resulted in Message: null
03:48:49.465 [task-scheduler-2] DEBUG o.s.i.e.SourcePollingChannelAdapter - \
Received no Message during the poll, returning 'false'
03:48:54.466 [task-scheduler-1] DEBUG o.s.i.e.SourcePollingChannelAdapter - \
Poll resulted in Message: null
03:48:54.467 [task-scheduler-1] DEBUG o.s.i.e.SourcePollingChannelAdapter - \
Received no Message during the poll, returning 'false'
```

In this log, we can see that the first time around the poller detects the one file that was in the directory and then afterward considers it processed, so the file inbound adapter does not process it a second time. There are additional options in `FsShellWritingMessageHandler` to enable the generation of an additional directory path that contains an embedded date or a UUID (universally unique identifier). To enable the output to have an additional dated directory path using the default path format (*year/month/day/hour/minute/second*), set the property `generateDestinationDirectory` to `true`. Setting `generateDestinationDirectory` to `true` would result in the file written into HDFS, as shown in Example 13-5.

Example 13-5. Partial output from running the file polling example with generateDestinationDirectory set to true

```
03:48:44.187 [main] INFO  c.o.s.hadoop.filepolling.FilePolling - \
File Polling Application Running
...
04:02:32.843 [task-scheduler-1] INFO  c.o.s.h.f.FsShellWritingMessageHandler - \
```

```
            sourceFile = /opt/application/logs/file_1.txt
04:02:32.843 [task-scheduler-1] INFO  c.o.s.h.f.FsShellWritingMessageHandler - \
            resultFile = /data/application/logs/2012/08/09/04/02/32/file_1.txt
```

Another way to move files into HDFS is to collect them via FTP from remote machines, as illustrated in Figure 13-4.

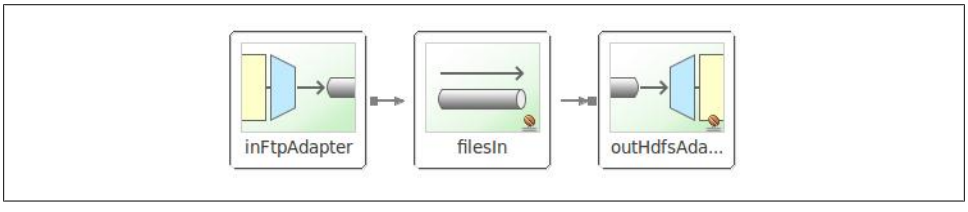


Figure 13-4. A Spring Integration data pipeline that polls an FTP site for files and copies them into HDFS

The configuration in Example 13-6 is similar to the one for file polling, only the configuration of the inbound adapter is changed.

Example 13-6. Defining a data pipeline that polls for files on an FTP site and loads them into HDFS

```
<context:property-placeholder location="ftp.properties,hadoop.properties"/>

<hdp:configuration>fs.default.name=${hd.fs}</hdp:configuration>

<bean id="ftpClientFactory"
      class="org.springframework.integration.ftp.session.DefaultFtpSessionFactory">
  <property name="host" value="${ftp.host}"/>
  <property name="port" value="${ftp.port}"/>
  <property name="username" value="${ftp.username}"/>
  <property name="password" value="${ftp.password}"/>
</bean>

<int:channel id="filesIn"/>

<int-ftp:inbound-channel-adapter id="inFtpAdapter"
    channel="filesIn"
    cache-sessions="false"
    session-factory="ftpClientFactory"
    filename-pattern="*.txt"
    auto-create-local-directory="true"
    delete-remote-files="false"
    remote-directory="${ftp.remoteDirectory}"
    local-directory="${ftp.localDirectory}">
  <int:poller fixed-rate="5000"/>
</int-ftp:inbound-channel-adapter>

<int:outbound-channel-adapter id="outHdfsAdapter"
    channel="filesIn" ref="fsShellWritingMessagingHandler"/>

<bean id="fsShellWritingMessagingHandler"
      class="com.oreilly.springdata.hadoop.ftp.FsShellWritingMessageHandler">
```

```
  <constructor-arg value="${ftp.destinationHdfsDirectory}"/>
  <constructor-arg ref="hadoopConfiguration"/>
</bean>
```

You can build and run this application using the commands shown in Example 13-7.

Example 13-7. Command to build and run the file polling example

```
$ cd hadoop/ftp
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/ftp
```

The configuration assumes there is a *testuser* account on the FTP host machine. Once you place a file in the outgoing FTP directory, you will see the data pipeline in action, copying the file to a local directory and then copying it into HDFS.

Event Streams

Streams are another common source of data that you might want to store into HDFS and optionally perform real-time analysis as it flows into the system. To meet this need, Spring Integration provides several inbound adapters that we can use to process streams of data. Once inside a Spring Integration, the data can be passed through a processing chain and stored into HDFS. The pipeline can also take parts of the stream and write data to other databases, both relational and NoSQL, in addition to forwarding the stream to other systems using one of the many outbound adapters. Figure 13-2 showed one example of this type of data pipeline. Next, we will use the TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) inbound adapters to consume data produced by syslog and then write the data into HDFS.

The configuration that sets up a TCP-syslog-to-HDFS processing chain is shown in Example 13-8.

Example 13-8. Defining a data pipeline that receives syslog data over TCP and loads it into HDFS

```
<context:property-placeholder location="hadoop.properties, syslog.properties"/>

<hdp:configuration register-url-handler="false">
  fs.default.name=${hd.fs}
</hdp:configuration>

<hdp:file-system id="hadoopFs"/>

<int-ip:tcp-connection-factory id="syslogListener"
                               type="server"
                               port="${syslog.tcp.port}"
                               deserializer="lfDeserializer"/>

<bean id="lfDeserializer"
      class="com.oreilly.springdata.integration.ip.syslog.ByteArrayLfSerializer"/>
```

```
<int-ip:tcp-inbound-channel-adapter id="tcpAdapter"
                                    channel="syslogChannel"
                                    connection-factory="syslogListener"/>

<!-- processing chain -->
<int:chain input-channel="syslogChannel">
  <int:transformer ref="sysLogToMapTransformer"/>
  <int:object-to-string-transformer/>
  <int:outbound-channel-adapter ref="hdfsWritingMessageHandler"/>
</int:chain>

<bean id="sysLogToMapTransformer"
      class="com.oreilly.springdata.integration.ip.syslog.SyslogToMapTransformer"/>

<bean id="hdfsWritingMessageHandler"
      class="com.oreilly.springdata.hadoop.streaming.HdfsWritingMessageHandler">
  <constructor-arg ref="hdfsWriterFactory"/>
</bean>

<bean id="hdfsWriterFactory"
      class="com.oreilly.springdata.hadoop.streaming.HdfsTextFileWriterFactory">
  <constructor-arg ref="hadoopFs"/>
  <property name="basePath" value="${syslog.hdfs.basePath}"/>
  <property name="baseFilename" value="${syslog.hdfs.baseFilename}"/>
  <property name="fileSuffix" value="${syslog.hdfs.fileSuffix}"/>
  <property name="rolloverThresholdInBytes"
      value="${syslog.hdfs.rolloverThresholdInBytes}"/>
</bean>
```

The relevant configuration parameters for the pipeline are externalized in the `streaming.properties` file, as shown in Example 13-9.

Example 13-9. The externalized properties for streaming data from syslog into HDFS

```
syslog.tcp.port=1514
syslog.udp.port=1513
syslog.hdfs.basePath=/data/
syslog.hdfs.baseFilename=syslog
syslog.hdfs.fileSuffix=log
syslog.hdfs.rolloverThresholdInBytes=500
```

The diagram for this data pipeline is shown in Figure 13-5.

This configuration will create a connection factory that listens for an incoming TCP connection on port 1514. The serializer segments the incoming byte stream based on the newline character in order to break up the incoming syslog stream into events. Note that this lower-level serializer configuration will be encapsulated in a syslog XML namespace in the future so as to simplify the configuration. The inbound channel adapter takes the syslog message off the TCP data stream and parses it into a byte array, which is set as the payload of the incoming message.

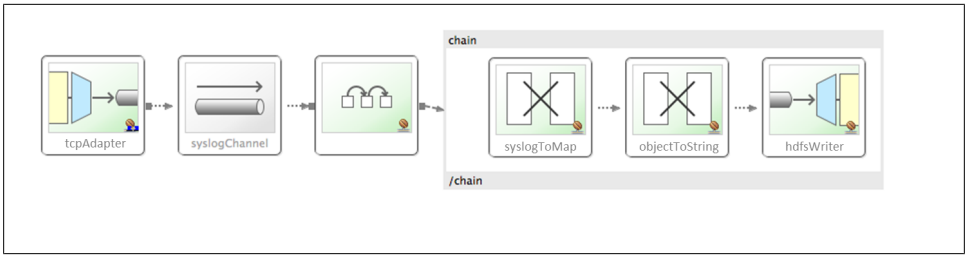


Figure 13-5. A Spring Integration data pipeline that streams data from syslog into HDFS

Spring Integration's chain component groups together a sequence of endpoints without our having to explicitly declare the channels that connect them. The first element in the chain parses the `byte[]` array and converts it to a `java.util.Map` containing the key/value pairs of the syslog message. At this stage, you could perform additional operations on the data, such as filtering, enrichment, real-time analysis, or routing to other databases. In this example, we have simply transformed the payload (now a `Map`) to a `String` using the built-in object-to-string transformer. This string is then passed into the `HdfsWritingMessageHandler` that writes the data into HDFS. `HdfsWritingMessageHandler` lets you configure the HDFS directory to write the files, the file naming policy, and the file size rollover policy. In this example, the rollover threshold was set artificially low (500 bytes versus the 10 MB default) to highlight the rollover capabilities in a simple test usage case.

To build and run this application, use the commands shown in Example 13-10.

Example 13-10. Commands to build and run the Syslog streaming example

```
$ cd hadoop/streaming
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/streaming
```

To send a test message, use the logger utility demonstrated in Example 13-11.

Example 13-11. Sending a message to syslog

```
$ logger -p local3.info -t TESTING "Test Syslog Message"
```

Since we set `HdfsWritingMessageHandler`'s `rolloverThresholdInBytes` property so low, after sending a few of these messages or just waiting for messages to come in from the operating system, you will see inside HDFS the files shown in Example 13-12.

Example 13-12. Syslog data in HDFS

```
$ hadoop dfs -ls /data
-rw-r--r-- 3 mpollack supergroup 711 2012-08-09 13:19 /data/syslog-0.log
-rw-r--r-- 3 mpollack supergroup 202 2012-08-09 13:22 /data/syslog-1.log
-rw-r--r-- 3 mpollack supergroup 240 2012-08-09 13:22 /data/syslog-2.log
-rw-r--r-- 3 mpollack supergroup 119 2012-08-09 15:04 /data/syslog-3.log
…
```

```
$ hadoop dfs -cat /data/syslog-2.log
{HOST=ubuntu, MESSAGE=Test Syslog Message, SEVERITY=6, FACILITY=19, \
 TIMESTAMP=Thu Aug 09 13:22:44 EDT 2012, TAG=TESTING}
{HOST=ubuntu, MESSAGE=Test Syslog Message, SEVERITY=6, FACILITY=19, \
 TIMESTAMP=Thu Aug 09 13:22:55 EDT 2012, TAG=TESTING}
```

To use UDP instead of TCP, remove the TCP-related definitions and add the commands shown in Example 13-13.

Example 13-13. Configuration to use UDP to consume syslog data

```
<int-ip:udp-inbound-channel-adapter id="udpAdapter"
        channel="syslogChannel" port="${syslog.udp.port}"/>
```

Event Forwarding

When you need to process a large amount of data from several different machines, it can be useful to forward the data from where it is produced to another server (as opposed to processing the data locally). The TCP inbound and outbound adapters can be paired together in an application so that they forward data from one server to another. The channel that connects the two adapters can be backed by several persistent message stores. Message stores are represented in Spring Integration by the interface `MessageStore`, and implementations are available for JDBC, Redis, MongoDB, and GemFire. Pairing inbound and outbound adapters together in an application affects the message processing flow such that the message is persisted in the message store of the producer application before the message is sent to the consumer application. The message is removed from the producer's message store once the acknowledgment from the consumer is received. The consumer sends its acknowledgment once it has successfully put the received message in its own message-store-backed channel. This configuration enables an additional level of "store and forward" guarantee via TCP normally found in messaging middleware such as JMS or RabbitMQ.

Example 13-14 is a simple demonstration of forwarding TCP traffic and using Spring's support to easily bootstrap an embedded HSQL database to serve as the message store.

Example 13-14. Store and forwarding of data across processes using TCP adapters

```
<int:channel id="dataChannel">
  <int:queue message-store="messageStore"/>
</int:channel>

<int-jdbc:message-store id="messageStore" data-source="dataSource"/>

<jdbc:embedded-database id="dataSource"/>

<int-ip:tcp-inbound-channel-adapter id="tcpInAdapter"
        channel="dataChannel" port="${syslog.tcp.in.port}"/>

<int-ip:tcp-outbound-channel-adapter id="tcpOutAdapter"
        channel="dataChannel"  port="${syslog.tcp.out.port}"/>
```

Management

Spring Integration provides two key features that let you manage data pipelines at runtime: the exporting of channels and endpoints to JMX and a control bus. Much like JMX, the control bus lets you invoke operations and view metric information related to each component, but it is more general-purpose because it allows you to run small programs inside the running application to change its state and behavior.

Exporting channels and endpoints to JMX is as simple as adding the lines of XML configuration shown in Example 13-15.

Example 13-15. Exporting channels and endpoints to JMX

```
<int-jmx:mbean-export default-domain="streaming-syslog"/>

<context:mbean-server/>
```

Running the TCP streaming example in the previous section and then starting JConsole shows the JMX metrics and operations that are available (Figure 13-6). Some examples are to start and stop the TCP adapter, and to get the min, max, and mean duration of processing in a `MessageHandler`.

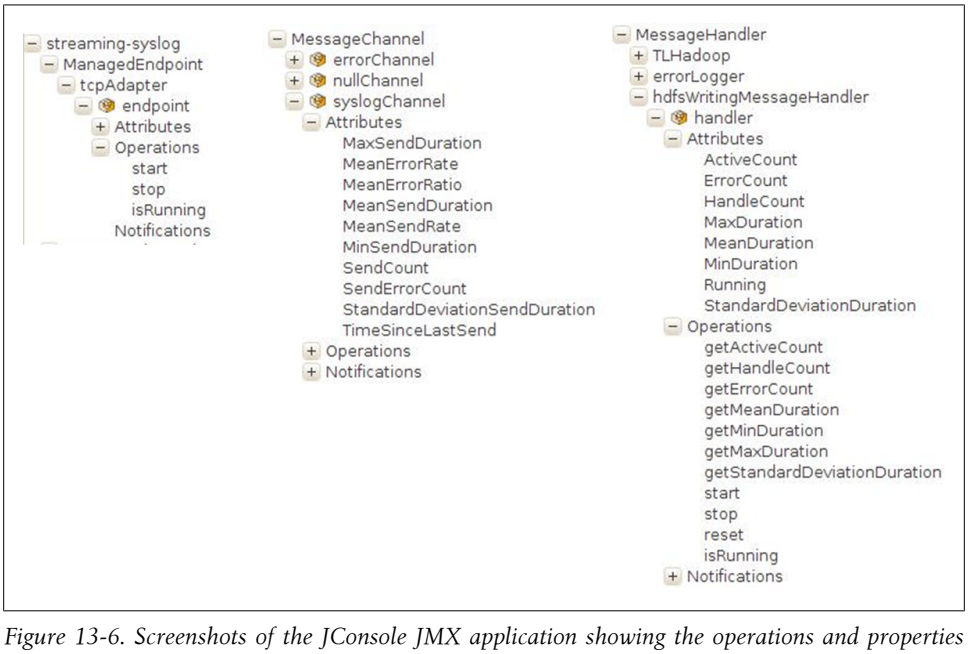


Figure 13-6. Screenshots of the JConsole JMX application showing the operations and properties available on the TcpAdapter, channels, and HdfsWritingMessageHandler

A control bus can execute Groovy scripts or Spring Expression Language (SpEL) expressions, allowing you to manipulate the state of components inside the application

programmatically. By default, Spring Integration exposes all of its components to be accessed through a control bus. The syntax for a SpEL expression that stops the TCP inbound adapter would be `@tcpAdapter.stop()`. The `@` prefix is an operator that will retrieve an object by name from the Spring `ApplicationContext`; in this case, the name is `tcpAdapter`, and the method to invoke is `stop`. A Groovy script to perform the same action would not have the `@` prefix. To declare a control bus, add the configuration shown in Example 13-16.

Example 13-16. Configuring a Groovy-based control bus

```
<int:channel id="inOperationChannel"/>

<int-groovy:control-bus input-channel="inOperationChannel"/>
```

By attaching an inbound channel adapter or gateway to the control bus's input channel, you can execute scripts remotely. It is also possible to create a Spring MVC application and have the controller send the message to the control bus's input channel, as shown in Example 13-17. This approach might be more natural if you want to provide additional web application functionality, such as security or additional views. Example 13-17 shows a Spring MVC controller that forwards the body of the incoming web request to the control bus and returns a `String`-based response.

Example 13-17. Spring MVC control to send message to the control bus

```
@Controller
public class ControlBusController {

  private @Autowired MessageChannel inOperationChannel;

  @RequestMapping("/admin")
  public @ResponseBody String simple(@RequestBody String message) {

    Message<String> operation = MessageBuilder.withPayload(message).build();
    MessagingTemplate template = new MessagingTemplate();
    Message response = template.sendAndReceive(inOperationChannel, operation);
    return response != null ? response.getPayload().toString() : null;

  }
}
```

Running the sample application again, we can interact with the control bus over HTTP using curl to query and modify the state of the inbound TCP adapter (Example 13-18).

Example 13-18. Configuring the control bus

```
$ cd streaming
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/streaming
… output omitted …
$ curl -X GET --data "tcpAdapter.isRunning()" http://localhost:8080/admin
true
$ curl -X GET --data "tcpAdapter.stop()" http://localhost:8080/admin
```

```
$ curl -X GET --data "tcpAdapter.isRunning()" http://localhost:8080/admin
false
$ curl -X GET --data "tcpAdapter.start()" http://localhost:8080/admin
$ curl -X GET --data "tcpAdapter.isRunning()" http://localhost:8080/admin
true
```

An Introduction to Spring Batch

The Spring Batch project was started in 2007 as a collaboration between SpringSource and Accenture to provide a comprehensive batch framework to support the development of robust batch applications. These batch applications require performing bulk processing of large amounts of data that are critical to the operation of a business. Spring Batch has been widely adopted and used in thousands of enterprise applications worldwide. Batch jobs have their own set of best practices and domain concepts, gathered over many years of building up Accenture's consulting business and encapsulated into the Spring Batch project. Thus, Spring Batch supports the processing of large volumes of data with features such as automatic retries after failure, skipping of records, job restarting from the point of last failure, periodic batch commits to a transactional database, reusable components (such as parsers, mappers, readers, processors, writers, and validators), and workflow definitions. As part of the Spring ecosystem, the Spring Batch project builds upon the core features of the Spring Framework, such as the use of the Spring Expression Language. Spring Batch also carries over the design philosophy of the Spring Framework, which emphasizes a POJO-based development approach and promotes the creation of maintainable, testable code.

The concept of a workflow in Spring Batch translates to a Spring Batch `Job` (not to be confused with a MapReduce job). A batch `Job` is a directed graph, each node of the graph being a processing `Step`. `Step`s can be executed sequentially or in parallel, depending on the configuration. `Job`s can be started, stopped, and restarted. Restarting jobs is possible since the progress of executed steps in a `Job` is persisted in a database via a `JobRepository`. Jobs are composable as well, so you can have a job of jobs. Figure 13-7 shows the basic components in a Spring Batch application. The `JobLauncher` is responsible for starting a job and is often triggered via a scheduler. The Spring Framework provides basic scheduling functionality as well as integration with Quartz; however, often enterprises use their own scheduling software such as Tivoli or Control-M. Other options to launch a job are through a RESTful administration API, a web application, or programmatically in response to an external event. In the latter case, using the Spring Integration project and its many channel adapters for communicating with external systems is a common choice. You can read more about combining Spring Integration and Spring Batch in the book *Spring Integration in Action* [Fisher12].For additional information on Spring Batch, consult the project website, which contains links to extensive reference documentation, sample applications, and links to several books.

The processing performed in a step is broken down into three stages—an `ItemReader`, `ItemProcessor`, and `ItemWriter`—as shown in Figure 13-8. Using an `ItemProcessor` is optional.

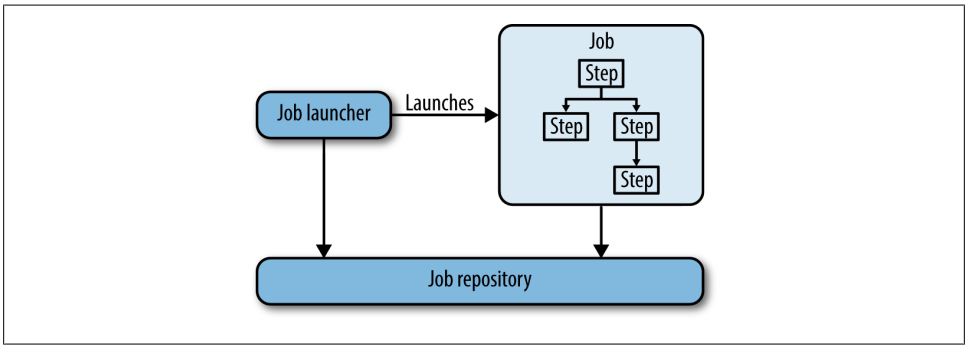


Figure 13-7. Spring Batch overview

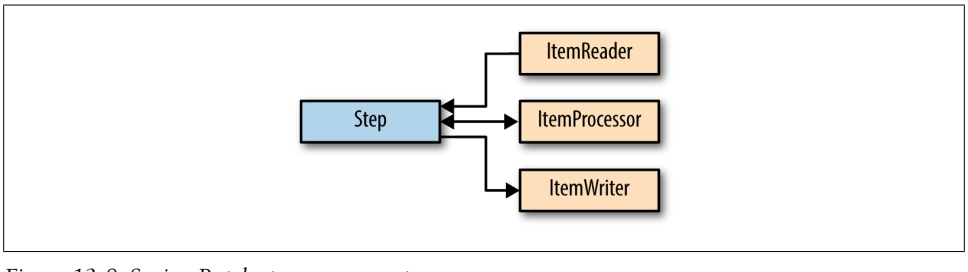


Figure 13-8. Spring Batch step components

One of the primary use cases for Spring Batch is to process the contents of large files and load the data into a relational database. In this case, a `FlatFileItemReader` and a `JdbcItemWriter` are used along with custom logic either configured declaratively or coded directly in an `ItemProcessor`. To increase the performance, the `Step` is "chunked," meaning that chunks of data, say 100 rows, are aggregated together and then passed to the `ItemWriter`. This allows us to efficiently process a group of records by using the batch APIs available in many databases to insert data. A snippet of configuration using the Spring Batch XML namespace that reads from a file and writes to a database is shown in Example 13-19. In subsequent sections, we will dive into the configuration of readers, writers, and processors.

Example 13-19. Configuring a Spring Batch step to process flat file data and copy it into a database

```
<step id="simpleStep">
  <tasklet>
    <chunk reader="flatFileItemReader" processor="itemProcessor" writer="jdbcItemWriter"
           commit-interval="100"/>
    </chunk>
```

```
  </tasklet>
</step>
```

Additional features available in Spring Batch allow you to scale up and out the job execution in order to handle the requirements of high-volume and high-performance batch jobs. For more information on these topics, refer to the Spring Batch reference guide or one of the Spring Batch books ([CoTeGreBa11], [Minella11]).

It is important to note that the execution model of a Spring Batch application takes place outside of the Hadoop cluster. Spring Batch applications can scale up by using different threads to concurrently process different files or scale out using Spring Batch's own master-slave remote partitioning model. In practice, scaling up with threads has been sufficient to meet the performance requirements of most users. You should try this option as a first strategy to scale before using remote partitioning. Another execution model that will be developed in the future is to run a Spring Batch job inside the Hadoop cluster itself, taking advantage of the cluster's resource management functionality to scale out processing across the nodes of the cluster, taking into account the locality of data stored in HDFS. Both models have their advantages, and performance isn't the only criteria to decide which execution model to use. Executing a batch job outside of the Hadoop cluster often enables easier data movement between different systems and multiple Hadoop clusters.

In the following sections, we will use the Spring Batch framework to process and load data into HDFS from a relational database. In the section "Exporting Data from HDFS" on page 243, we will export data from HDFS into a relational database and the MongoDB document database.

Processing and Loading Data from a Database

To process and load data from a relational database to HDFS, we need to configure a Spring Batch tasklet with a `JdbcItemReader` and a `HdfsTextItemWriter`. The sample application for this section is located in *./hadoop/batch-import* and is based on the sample code that comes from the book *Spring Batch in Action*. The domain for the sample application is an online store that needs to maintain a catalog of the products it sells. We have modified the example only slightly to write to HDFS instead of a flat file system. The configuration of the Spring Batch tasklet is shown in Example 13-20.

Example 13-20. Configuring a Spring Batch step to read from a database and write to HDFS

```
<job id="importProducts" xmlns="http://www.springframework.org/schema/batch">
  <step id="readWriteProducts">
    <tasklet>
      <chunk reader="jdbcReader" writer="hdfsWriter" commit-interval="100"/>
   </tasklet>
  </step>
</job>

<bean id="jdbcReader" class="org.springframework.batch.item.database.JdbcCursorItemReader">
```

```
  <property name="dataSource" ref="dataSource"/>
  <property name="sql" value="select id, name, description, price from product"/>
  <property name="rowMapper" ref="productRowMapper"/>
</bean>

<bean id="productRowMapper" class="com.oreilly.springdata.domain.ProductRowMapper"/>
```

We configure the `JdbcCursorItemReader` with a standard JDBC `DataSource` along with the SQL statement that will select the data from the product table that will be loaded into HDFS. To start and initialize the database with sample data, run the commands shown in Example 13-21.

Example 13-21. Commands to initalize and run the H2 database for a Spring Batch application

```
$ cd hadoop/batch-import
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/start-database
```

A browser will pop up that lets you browse the contents of the database containing the product table in addition to the tables for Spring Batch used to implement the job repository.

The commit interval is set to 100, which is more than the amount of data available in this simple application, but represents a typical number to use. For each 100 records read from the database, the transaction that updates the job execution metadata will be committed to the database. This allows for a restart of the job upon a failure to pick up where it left off.

The `rowMapper` property of `JdbcCursorItemReader` is an implementation of Spring's `RowMapper` interface, which is part of Spring's JDBC feature set. The `RowMapper` interface provides a simple way to convert a JDBC `ResultSet` to a POJO when a single row in a `ResultSet` maps onto a single POJO instance. Iteration over the `ResultSet` as well as exception handling (which is normally quite verbose and error-prone) is encapsulated by Spring, letting you focus on writing only the required mapping code. The `ProductRowMapper` used in this application converts each row of the `ResultSet` object to a `Product` Java object and is shown in Example 13-22. The `Product` class is a simple POJO with getters and setters that correspond to the columns selected from the product table.

Example 13-22. The ProductRowMapper that converts a row in a ResultSet to a Product object

```
public class ProductRowMapper implements RowMapper<Product> {

  public Product mapRow(ResultSet rs, int rowNum) throws SQLException {
    Product product = new Product();
    product.setId(rs.getString("id"));
    product.setName(rs.getString("name"));
    product.setDescription(rs.getString("description"));
    product.setPrice(rs.getBigDecimal("price"));
    return product;
  }
}
```

The `JdbcCursorItemReader` class relies on the streaming functionality of the underlying JDBC driver to iterate through the result set in an efficient manner. You can set the property `fetchSize` to give a hint to the driver to load only a certain amount of data into the driver that runs in the client process. The value to set the `fetchSize` to depends on the JDBC driver. For example, in the case of MySQL, the documentation suggests setting `fetchSize` to `Integer.MIN_VALUE`, a nonobvious choice for handling large result sets efficiently. Of note, Spring Batch also provides the class `JdbcPagingItemReader` as another strategy to control how much data is loaded from the database into the client process as well as the ability to load data from stored procedures.

The last part of the configuration for the application to run is the `hdfsWriter` shown in Example 13-23.

Example 13-23. The configuration of the HdfsTextItemWriter

```
<context:property-placeholder location="hadoop.properties"/>

<hdp:configuration>fs.default.name=${hd.fs}</hdp:configuration>
<hdp:file-system id="hadoopFs"/>

<bean id="hdfsWriter" class="com.oreilly.springdata.batch.item.HdfsTextItemWriter">
  <constructor-arg ref="hadoopFs"/>
  <property name="basePath" value="/import/data/products/"/>
  <property name="baseFilename" value="product"/>
  <property name="fileSuffix" value="txt"/>
  <property name="rolloverThresholdInBytes" value="100"/>
  <property name="lineAggregator">
   <bean class="org.springframework.batch.item.file.transform.PassThroughLineAggregator"/>
  </property>
</bean>
```

The Hadoop configuration is as we have seen in the previous sections, but the `<hdp:file-system/>` is new. It is responsible for creating the appropriate `org.apache.hadoop.fs.FileSystem` implementation based on the Hadoop configuration. The possible options are implementations that communicate with HDFS using the standard HDFS protocol (*hdfs://*), HFTP (*hftp://*), or WebHDFS (*webhdfs://*). The `FileSystem` is used by `HdfsTextItemWriter` to write plain-text files to HDFS. The configuration of the `HdfsTextItemWriter`'s properties—`basePath`, `baseFileName`, and `fileSuffix`—will result in files being written into the */import/data/products* directory with names such as *product-0.txt* and *product-1.txt*. We set `rolloverThresholdInBytes` to a very low value for the purposes of demonstrating the rollover behavior.

`ItemWriters` often require a collaborating object, an implementation of the `LineAggregator` interface that is responsible for converting the item being processed into a string. In this example, we are using the `PassThroughFieldExtractor` provided by Spring Batch, which will delegate to the `toString()` method of the `Product` class to create the string. The `toString()` method of `Product` is a simple comma-delimited concatenation of the ID, name, description, and price values.

To run the application and import from a database to HDFS, execute the commands shown in Example 13-24.

Example 13-24. Commands to import data from a database to HDFS

```
$ cd hadoop/batch-import
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/import
```

Example 13-25 shows the resulting content in HDFS.

Example 13-25. The imported product data in HDFS

```
$ hadoop dfs -ls /import/data/products

Found 6 items
-rw-r--r-- 3 mpollack supergroup 114 2012-08-21 11:40 /import/data/products/product-0.txt
-rw-r--r-- 3 mpollack supergroup 113 2012-08-21 11:40 /import/data/products/product-1.txt
-rw-r--r-- 3 mpollack supergroup 122 2012-08-21 11:40 /import/data/products/product-2.txt
-rw-r--r-- 3 mpollack supergroup 119 2012-08-21 11:40 /import/data/products/product-3.txt
-rw-r--r-- 3 mpollack supergroup 136 2012-08-21 11:40 /import/data/products/product-4.txt
-rw-r--r-- 3 mpollack supergroup  51 2012-08-21 11:40 /import/data/products/product-5.txt

$ hadoop dfs -cat /import/data/products/

PR1...210,BlackBerry 8100 Pearl,,124.6
PR1...211,Sony Ericsson W810i,,139.45
PR1...212,Samsung MM-A900M Ace,,97.8
```

There are many other `LineAggregator` implementations available in Spring Batch that give you a great deal of declarative control over what fields are written to the file and what characters are used to delimit each field. Example 13-26 shows one such implementation.

Example 13-26. Specifying JavaBean property names to create the String written to HDFS for each product object

```
<property name="lineAggregator">
 <bean class="org.springframework.batch.item.file.transform.DelimitedLineAggregator">
  <property name="fieldExtractor">
   <bean class="org.springframework.batch.item.file.transform.BeanWrapperFieldExtractor">
    <property name="names" value="id,price,name"/>
   </bean>
  </property>
 </bean>
</property>
```

The `DelimitedLineAggregator` will use a comma to separate fields by default, and the `BeanWrapperFieldExtractor` is passed in the JavaBean property names that it will select from the `Product` object to write a line of output to HDFS.

Running the application again with this configuration of a `LineAggregator` will create files in HDFS that have the content shown in Example 13-27.

Example 13-27. The imported product data in HDFS using an alternative formatting

```
$ hadoop dfs -cat /import/data/products/products-0.txt

PR1...210,124.6,BlackBerry 8100 Pearl
PR1...211,139.45,Sony Ericsson W810i
PR1...212,97.8,Samsung MM-A900M Ace
```

Hadoop Workflows

Hadoop applications rarely consist of one MapReduce job or Hive script. Analysis logic is usually broken up into several steps that are composed together into a chain of execution to perform the complete analysis task. In the previous MapReduce and Apache weblog examples, we used `JobRunner`s, `HiveRunner`s, and `PigRunner`s to execute HDFS operations and MapReduce, Hive, or Pig jobs, but that is not a completely satisfactory solution. As the number of steps in an analysis chain increases, the flow of execution is hard to visualize and not naturally structured as a graph structure in the XML namespace. There is also no tracking of the execution steps in the analysis chain when we're using the various Runner classes. This means that if one step in the chain fails, we must restart it (manually) from the beginning, making the overall "wall clock" time for the analysis task significantly larger as well as inefficient. In this section, we will introduce extensions to the Spring Batch project that will provide structure for chaining together multiple Hadoop operations into what are loosely called *workflows*.

Spring Batch Support for Hadoop

Because Hadoop is a batch-oriented system, Spring Batch's domain concepts and workflow provide a strong foundation to structure Hadoop-based analysis tasks. To make Spring Batch "Hadoop aware," we take advantage of the fact that the processing actions that compose a Spring Batch `Step` are pluggable. The plug-in point for a `Step` is known as a `Tasklet`. Spring for Apache Hadoop provides custom `Tasklet`s for HDFS operations as well as for all types of Hadoop jobs: MapReduce, Streaming, Hive, and Pig. This allows for creating workflows as shown in Figure 13-9.

There is support in the Eclipse-based Spring Tool Suite (STS) to support the visual authoring of Spring Batch jobs. Figure 13-10 shows the equivalent diagram to Figure 13-9 inside of STS.

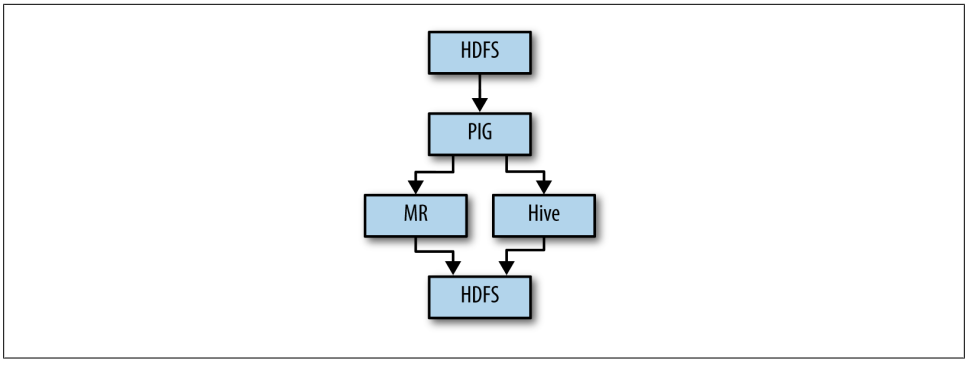


Figure 13-9. Steps in a Spring Batch application that execute Hadoop HDFS operations and run Pig, MapReduce, and Hive jobs

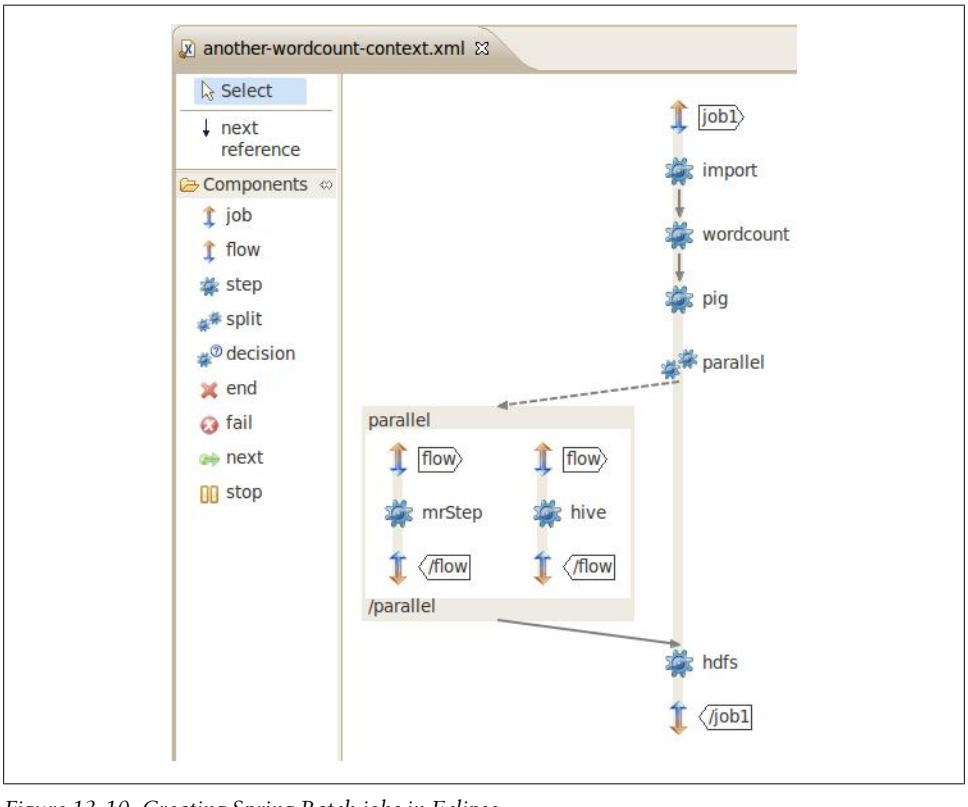


Figure 13-10. Creating Spring Batch jobs in Eclipse

The underlying XML for a Spring Batch job with a linear flow of steps has the general pattern shown in Example 13-28.

Example 13-28. Spring Batch job definition for a sequence of steps

```
<job id="job">
    <step id="stepA" next="stepB"/>
    <step id="stepB" next="stepC"/>
    <step id="stepC"/>
</job>
```

You can also make the flow conditional based on the exit status of the step. There are several well-known `ExitStatus` codes that a step returns, the most common of which are `COMPLETED` and `FAILED`. To create a conditional flow, you use the nested `next` element of the step, as shown in Example 13-29.

Example 13-29. Spring Batch job definition for a conditional sequence of steps

```
<job id="job">
    <step id="stepA">
        <next on="FAILED" to="stepC"/>
        <next on="*" to="stepB"/>
    </step>
    <step id="stepB" next="stepC"/>
    <step id="stepC"/>
</job>
```

In this example, if the exit code matches `FAILED`, the next step executed is `stepC`; otherwise, `stepB` is executed followed by `stepC`.

There is a wide range of ways to configure the flow of a Spring that we will not cover in this section. To learn more about how to configure more advanced job flows, see the reference documentation or one of the aforementioned Spring Batch books.

To configure a Hadoop-related step, you can use the XML namespace provided by Spring for Apache Hadoop. Next, we'll show how we can configure the wordcount example as a Spring Batch application, reusing the existing configuration of a MapReduce and HDFS script that were part of the standalone Hadoop application examples used previously. Then we will show how to configure other Hadoop-related steps, such as for Hive and Pig.

Wordcount as a Spring Batch Application

The wordcount example has two steps: importing data into HDFS and then running a MapReduce job. Example 13-30 shows the Spring Batch job representing the workflow using the Spring Batch XML namespace. We use the namespace prefix `batch` to distinguish the batch configuration from the Hadoop configuration.

Example 13-30. Setting up the Spring Batch job to perform HDFS and MapReduce steps

```
<batch:job id="job1">
  <batch:step id="import" next="wordcount">
    <batch:tasklet ref="scriptTasklet"/>
  </batch:step>

  <batch:step id="wordcount">
    <batch:tasklet ref="wordcountTasklet"/>
  </batch:step>
</batch:job>

<tasklet id="wordcountTasklet" job-ref="wordcountJob"/>
<script-tasklet id="scriptTasklet" script-ref="hdfsScript">

<!-- MapReduce job and HDFS script as defined in previous examples -->
<job id="wordcountJob"
     input-path="${wordcount.input.path}"
     output-path="${wordcount.output.path}"
     mapper="org.apache.hadoop.examples.WordCount.TokenizerMapper"
     reducer="org.apache.hadoop.examples.WordCount.IntSumReducer"/>

<script id="hdfsScript" location="copy-data.groovy" >
  <property name="inputPath" value="${wordcount.input.path}"/>
  <property name="outputPath" value="${wordcount.output.path}"/>
  <property name="localResource" value="${local.data}"/>
</script>
```

It is common to parameterize a batch application by providing job parameters that are passed in when the job is launched. To change the batch job to reference these job parameters instead of ones that comes from static property files, we need to make a few changes to the configuration. We can retrieve batch job parameters using SpEL, much like how we currently reference variables using `${...}` syntax.

The syntax of SpEL is similar to Java, and expressions are generally just one line of code that gets evaluated. Instead of using the syntax `${...}` to reference a variable, use the syntax `#{...}` to evaluate an expression. To access the Spring Batch job parameters, we'd use the expression `#{jobParameters['mr.input']}`. The variable `jobParameters` is available by default when a bean is placed in Spring Batch's step scope. The configuration of the MapReduce job and HDFS script, then, looks like Example 13-31.

Example 13-31. Linking the Spring Batch Tasklets to Hadoop Job and Script components

```
<job id="wordcount-job" scope="step"
     input-path="#{jobParameters['mr.input']}"
     output-path="#{jobParameters['mr.output']}"
     mapper="org.apache.hadoop.examples.WordCount.TokenizerMapper"
     reducer="org.apache.hadoop.examples.WordCount.IntSumReducer"/>

<script id="hdfsScript" location="copy-files.groovy" scope="step">
  <property name="localSourceFile" value="#{jobParameters['localData']}"/>
  <property name="hdfsInputDir" value="#{jobParameters['mr.input']}"/>
```

```
  <property name="hdfsOutputDir" value="#{jobParameters['mr.output']}"/>
</script>
```

The main application that runs the batch application passes in values for these parameters, but we could also set them using other ways to launch a Spring Batch application. The `CommandLineJobRunner`, administrative REST API, or the administrative web application are common choices. Example 13-32 is the main driver class for the sample application.

Example 13-32. Main application that launched a batch job

```
ApplicationContext context =
  new ClassPathXmlApplicationContext("classpath:/META-INF/spring/*-context.xml");
JobLauncher jobLauncher = context.getBean(JobLauncher.class);
Job job = context.getBean(Job.class);
jobLauncher.run(job, new JobParametersBuilder()
  .addString("mr.input", "/user/gutenberg/input/word/")
  .addString("mr.output", "/user/gutenberg/output/word/")
  .addString("localData", "./data/nietzsche-chapter-1.txt")
  .addDate("date", new Date()).toJobParameters());
```

To run the batch application, execute the commands shown in Example 13-33.

Example 13-33. Commands to run the wordcount batch application

```
$ cd hadoop/batch-wordcount
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/batch-wordcount
```

Hive and Pig Steps

To execute a Hive script as part of a Spring Batch workflow, use the Hive `Tasklet` element, as shown in Example 13-34.

Example 13-34. Configuring a Hive Tasklet

```
<job id="job1" xmlns="http://www.springframework.org/schema/batch">
  <step id="import" next="hive">
    <tasklet ref="scriptTasklet"/>
  </step>

  <step id="hive">
    <tasklet ref="hiveTasklet"/>
  </step>
</job>

<hdp:hive-client-factory host="${hive.host}" port="${hive.port}"/>

<hive-tasklet id="hiveTasklet">
   <hdp:script location="analysis.hsql"/>
</hdp:hive-tasklet>
```

To execute a Pig script as part of a Spring Batch workflow, use the Pig `Tasklet` element (Example 13-35).

Example 13-35. Configuring a Pig Tasklet

```
<job id="job1" xmlns="http://www.springframework.org/schema/batch">
  <step id="import" next="pig">
    <tasklet ref="scriptTasklet"/>
  </step>

  <step id="pig">
    <tasklet ref="pigTasklet"/>
  </step>
</job>

<pig-factory/>

<pig-tasklet id="pigTasklet">
  <script location="analysis.pig">
    <arguments>
      piggybanklib=${pig.piggybanklib}
      inputPath=${pig.inputPath}
      outputPath=${pig.outputPath}
    </arguments>
  </script>
</pig-tasklet>
```

Exporting Data from HDFS

The results from data analysis in Hadoop are often copied into structured data stores, such as a relational or NoSQL database, for presentation purposes or further analysis. One of the main use cases for Spring Batch is moving data back between files and databases and processing it along the way. In this section, we will use Spring Batch to export data from HDFS, perform some basic processing on the data, and then store the data outside of HDFS. The target data stores are a relational database and MongoDB.

From HDFS to JDBC

Moving the result data created from MapReduce jobs located in HDFS into a relational database is very common. Spring Batch provides many out-of-the-box components that you can configure to perform this activity. The sample application for this section is located in *./hadoop/batch-extract* and is based on the sample code that comes from the book *Spring Batch in Action*. The domain for the sample application is an online store that needs to maintain a catalog of the products it sells. The application as it was originally written reads product data from flat files on a local filesystem and then writes the product data into a product table in a relational database. We have modified the example to read from HDFS and also added error handling to show an additional Spring Batch feature.

To read from HDFS instead of a local filesystem, we need to register a `HdfsResourceLoader` with Spring to read data from HDFS using Spring's `Resource` abstraction. This lets us use the Spring Batch's existing `FlatFileItemReader` class, as it is based on the `Resource` abstraction. Spring's `Resource` abstraction provides a uniform way to read an `InputStream` from a variety of sources such as a URL (*http*, *ftp*), the Java `ClassPath`, or the standard filesystem. The `Resource` abstraction also supports reading from multiple source locations through the use of Ant-style regular expressions.To configure Spring to be able to read from HDFS as well as make HDFS the default resource type that will be used (e.g., *hdfs://hostname:port* versus *file://*), add the lines of XML shown in Example 13-36 to a Spring configuration file.

Example 13-36. Configuring the default resource loader to use HDFS

```
<context:property-placeholder location="hadoop.properties"/>

<hdp:configuration>fs.default.name=${hd.fs}</hdp:configuration>
<hdp:resource-loader id="hadoopResourceLoader"/>

<bean id="defaultResourceLoader"
  class="org.springframework.data.hadoop.fs.CustomResourceLoaderRegistrar">
  <property name="loader" ref="hadoopResourceLoader"/>
</bean>
```

The basic concepts of Spring Batch—such as jobs, steps, `ItemReader`, processors, and writers—were explained in "An Introduction to Spring Batch" on page 232. In this section, we will configure these components and discuss some of their configuration properties. However, we can't go into detail on all the ways to configure and run Spring Batch applications; there is a great deal of richness in Spring Batch relating to error handling, notifications, data validation, data processing, and scaling that we simply can't cover here. For additional information, you should consult the Spring Reference manual or one of the books on Spring Batch mentioned earlier.

Example 13-37 is the top-level configuration to create a Spring Batch job with a single step that processes the output files of a MapReduce job in HDFS and writes them to a database.

Example 13-37. Configuration of a Spring Batch job to read from HDFS and write to a relational database

```
<job id="exportProducts">
  <step id="readWriteProducts">
    <tasklet>
      <chunk reader="hdfsReader" processor="processor" writer="jdbcWriter"
             commit-interval="100" skip-limit="5">
        <skippable-exception-classes>
          <include class="org.springframework.batch.item.file.FlatFileParseException"/>
        </skippable-exception-classes>
      </chunk>
      <listeners>
        <listener ref="jdbcSkipListener"/>
      </listeners>
```

```
    </tasklet>
  </step>
</job>
```

The job defines only one step, which contains a reader, processor, and a writer. The commit interval refers to the number of items to process and aggregate before committing them to the database. In practice, the commit interval value is varied to determine which value will result in the highest performance. Values between 10 and a few hundred are what you can expect to use for this property. One of Spring Batch's features relating to resilient handling of errors is shown here: the use of the `skip-limit` and `skippable-exception-classes` properties. These properties determine how many times a specific error in processing will be allowed to occur before failing the step. The `skip-limit` determines how many times an exception can be thrown before failing the job. In this case, we are tolerating up to five throws of a `FlatFileParseException`. To keep track of all the lines that were not processed correctly, we configure a listener that will write the offending data into a separate database table. The listener extends the Spring Batch class `SkipListenerSupport`, and we override the `onSkipInRead(Throwable t)` that provides information on the failed import line.

Since we are going to read many files created from a MapReduce job (e.g., part-r-00001 and part-r-00002), we use Spring Batch's `MultiResourceItemReader`, passing in the HDFS directory name as a job parameter and a reference to a `FlatFileItemReader` that does the actual work of reading individual files from HDFS. See Example 13-38.

Example 13-38. Configuration of a Spring Batch HDFS reader

```
<bean id="hdfsReader"
  class="org.springframework.batch.item.file.MultiResourceItemReader" scope="step">
  <property name="resources" value="#{jobParameters['hdfsSourceDirectory']}"/>
  <property name="delegate" ref="flatFileItemReader"/>
</bean>

<bean id="flatFileItemReader"
       class="org.springframework.batch.item.file.FlatFileItemReader">
  <property name="lineMapper">
    <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">
      <property name="lineTokenizer">
        <bean
          class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
          <property name="names" value="id, name, description, price"/>
        </bean>
      </property>
      <property name="fieldSetMapper">
        <bean class="com.oreilly.springdata.batch.item.file.ProductFieldSetMapper"/>
      </property>
   </bean>
  </property>
</bean>
```

When launching the job, we provide the job parameter named `hdfsSourceDirectory` either programmatically, or through the REST API/web application if using Spring

Batch's administration features. Setting the scope of the bean to `step` enables resolution of `jobParameter` variables. Example 13-39 uses a main Java class to load the Spring Batch configuration and launch the job.

Example 13-39. Launching a Spring Batch job with parameters

```
public static void main(String[] args) throws Exception {

  ApplicationContext ctx =
    new ClassPathXmlApplicationContext("classpath*:/META-INF/spring/*.xml")
  JobLauncher jobLauncher = ctx.getBean(JobLauncher.class);
  Job job = ctx.getBean(Job.class);

  jobLauncher.run(job, new JobParametersBuilder()
    .addString("hdfsSourceDirectory", "/data/analysis/results/part-*")
    .addDate("date", new Date())
    .toJobParameters());
}
```

The `MultiResourceItemReader` class is what allows multiple files to be processed from the HDFS directory */data/analysis/results* that have a filename matching the expression `part-*`. Each file that is found by the `MultiResourceItemReader` is processed by the `FlatFileItemReader`. A sample of the content in each file is shown in Example 13-40. There are four columns in this data, representing the product ID, name, description, and price. (The description is empty in the sample data.)

Example 13-40. Sample content of HDFS file being exported into a database

```
PR1...210,BlackBerry 8100 Pearl,,124.60cl
PR1...211,Sony Ericsson W810i,,139.45
PR1...212,Samsung MM-A900M Ace,,97.80
PR1...213,Toshiba M285-E 14,,166.20
PR1...214,Nokia 2610 Phone,,145.50
…
PR2...315,Sony BDP-S590 3D Blu-ray Disk Player,,86.99
PR2...316,GoPro HD HERO2,,241.14
PR2...317,Toshiba 32C120U 32-Inch LCD HDTV,,239.99
```

We specify `FlatFileItemReader`'s functionality by configuring two collaborating objects of the `DefaultLineMapper`. The first is the `DelimitedLineTokenizer` provided by Spring Batch, which reads a line of input, and by default, tokenizes it based on commas. The field names and each value of the token are placed into Spring Batch's `FieldSet` object. The `FieldSet` object is similar to a JDBC result set but for data read from files. It allows you to access fields by name or position and to convert the values of those fields to Java types such as `String`, `Integer`, or `BigDecimal`. The second collaborating object is the `ProductFieldSetMapper`, provided by us, which converts the `FieldSet` to a custom domain object, in this case the `Product` class.

The `ProductFieldSetMapper` is extremely similar to the `ProductRowMapper` used in the previous section when reading from the database and writing to HDFS. See Example 13-41.

Example 13-41. Converting a FieldSet to a Product domain object

```
public class ProductFieldSetMapper implements FieldSetMapper<Product> {

  public Product mapFieldSet(FieldSet fieldSet) {
    Product product = new Product();
    product.setId(fieldSet.readString("id"));
    product.setName(fieldSet.readString("name"));
    product.setDescription(fieldSet.readString("description"));
    product.setPrice(fieldSet.readBigDecimal("price"));
    return product;
  }
}
```

The two parts that remain to be configured are the `ItemProcessor` and `ItemWriter`; they are shown in Example 13-42.

Example 13-42. Configuration of a Spring Batch item process and JDBC writer

```
<bean id="processor" class="com.oreilly.springdata.batch.item.ProductProcessor"/>

<bean id="jdbcWriter" class="org.springframework.batch.item.database.JdbcBatchItemWriter">
  <property name="dataSource" ref="dataSource"/>
  <property name="sql"
            value="INSERT INTO PRODUCT (ID, NAME, PRICE) VALUES (:id, :name, :price)"/>
  <property name="itemSqlParameterSourceProvider">
    <bean class="org.sfw.batch.item.database.BeanPropertyItemSqlParameterSourceProvider"/>
  </property>
</bean>
```

`ItemProcessor`s are commonly used to transform, filter, or validate the data. In Example 13-43, we use a simple filter that will pass only through records whose product description ID starts with PR1. Returning a `null` value from an `ItemProcessor` is the contract to filter out the item. Note that if you do not want to perform any processing and directly copy the input file to the database, you can simply remove the processor attribute from the tasklet's chunk XML configuration.

Example 13-43. A simple filtering ItemProcessor

```
public class ProductProcessor implements ItemProcessor<Product, Product> {

  @Override
  public Product process(Product product) throws Exception {
    if (product.getId().startsWith("PR1")) {
      return null;
    } else {
      return product;
    }
  }
}
```

The `JdbcBatchItemWriter` groups together a batch of SQL statements to commit together to the database. The batch size is equal to the commit interval defined previously.

We connect to the database using a standard JDBC `DataSource`, and the SQL statement is specified inline. What is nice about the SQL statement is that we can use named parameters instead of positional `?` placeholders. This is a feature provided by `BeanPropertyItemSqlParameterSourceProvider`, which associates the names of properties of the `Product` object with the `:name` values inside the SQL statement. Example 13-44 uses the H2 database and the schema for the product table.

Example 13-44. Schema definition of the product table

```
create table product (
  id character(9) not null,
  name character varying(50),
  description character varying(255),
  price float,
  update_timestamp timestamp,
  constraint product_pkey primary key (id)
);
```

To start the database, copy the sample data into HDFS, create the Spring Batch schema, and execute the commands shown in Example 13-45. Running these commands will also launch the H2 interactive web console.

Example 13-45. Building the example and starting the database

```
$ cd hadoop/batch-extract
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/start-database &
```

Next run the export process, as shown in Example 13-46.

Example 13-46. Running the export job

```
$ sh ./target/appassembler/bin/export
INFO - Loaded JDBC driver: org.h2.Driver
INFO - Established shared JDBC Connection: conn0: \
url=jdbc:h2:tcp://localhost/mem:hadoop_export user=SA
INFO - No database type set, using meta data indicating: H2
INFO - No TaskExecutor has been set, defaulting to synchronous executor.
INFO - Job: [FlowJob: [name=exportProducts]] launched with the following parameters: \
       [{hdfsSourceDirectory=/data/analysis/results/part-*, date=1345578343793}]
INFO - Executing step: [readWriteProducts]
INFO - Job: [FlowJob: [name=exportProducts]] completed with the following parameters: \
       [{hdfsSourceDirectory=/data/analysis/results/part-*, date=1345578343793}] and the \
       following status: [COMPLETED]
```

We can view the imported data using the H2 web console. Spring Batch also has an administrative console, where you can browse what jobs are available to be run, as well as look at the status of each job execution. To launch the administrative console, run **`sh ./target/appassembler/bin/launchSpringBatchAdmin`**, open a browser to *http://localhost:8080/springbatchadmin/jobs/executions*, and select the recent job execution link from the table. By clicking on the link for a specific job execution, you can view details about the state of the job.

From HDFS to MongoDB

To write to MongoDB instead of a relational database, we need to change the `ItemWriter` implementation from `JdbcBatchItemWriter` to `MongoItemWriter`. Example 13-47 shows a simple implementation of `MongoItemWriter` that writes the list of items using MongoDB's batch functionality, which inserts the items into the collection by making a single call to the database.

Example 13-47. An ItemWriter implementation that writes to MongoDB

```
public class MongoItemWriter implements ItemWriter<Object> {

  private MongoOperations mongoOperations;
  private String collectionName = "/data";

  // constructor and setters omitted.

  @Override
  public void write(List<? extends Object> items) throws Exception {
    mongoOperations.insert(items, collectionName);
  }
}
```

Spring's `MongoTemplate` (which implements the interface `MongoOperations`) provides support for converting Java classes to MongoDB's internal data structure format, `DbObject`. We can specify the connectivity to MongoDB using the Mongo XML namespace. Example 13-48 shows the configuration of `MongoItemWriter` and its underlying dependencies to connect to MongoDB.

Example 13-48. Configuration of a Spring Batch job to read from HDFS and write to MongoDB

```
<job id="exportProducts">
  <step id="readWriteProducts">
    <tasklet>
      <chunk reader="reader" writer="mongoWriter" commit-interval="100" skip-limit="5">
        <skippable-exception-classes>
          <include class="org.springframework.batch.item.file.FlatFileParseException"/>
        </skippable-exception-classes>
      </chunk>
    </tasklet>
  </step>
</job>

<!-- reader configuration is the same as before -->

<bean id="mongoWriter" class="com.oreilly.springdata.batch.item.mongodb.MongoItemWriter">
  <constructor-arg ref="mongoTemplate"/>
  <property name="collectionName" value="products"/>
</bean>

<bean id="mongoTemplate" class="org.springframework.data.mongodb.core.MongoTemplate">
    <constructor-arg ref="mongo"/>
    <constructor-arg name="databaseName" value="test"/>
```

```
</bean>

<mongo:mongo host="localhost" port="27017"/>
```

Running the application again will produce the contents of the products collection in the test database, shown in Example 13-49.

Example 13-49. Viewing the exported products collection in the MongoDB shell

```
$ mongo
> use test
> db.products.find()
{"_id" : "PR 210", "PRICE" : "124.60", "NAME" : "BlackBerry 8100 Pearl",
    "DESCRIPTION" : ""}
{"_id" : "PR 211", "PRICE" : "139.45", "NAME" : "Sony Ericsson W810i", "DESCRIPTION" : ""}
{"_id" : "PR 212", "PRICE" : "97.80", "NAME" : "Samsung MM-A900M Ace", "DESCRIPTION" : ""}

(output truncated)
```

This example shows how various Spring Data projects build upon each other to create important new features with a minimal amount of code. Following the same pattern to implement a `MongoItemWriter`, you can also easily create `ItemWriter`s for Redis or GemFire that would be as simple as the code shown in this section.

Collecting and Loading Data into Splunk

Splunk collects, indexes, searches, and monitors large amounts of machine-generated data. Splunk can process streams of real-time data as well as historical data. The first version of Splunk was released in 2005, with a focus on analyzing the data generated inside of datacenters to help solve operational infrastructure problems. As such, one of its core capabilities is moving data generated on individual machines into a central repository, as well as having out-of-the box knowledge of popular log file formats and infrastructure software like syslog. Splunk's base architecture consists of a splunkd daemon that processes and indexes streaming data, and a web application that allows users to search and create reports. Splunk can scale out by adding separate indexer, search, and forwarder instances as your data requirements grow. For more details on how to install, run, and develop with Splunk, refer to the product website, as well as the book *Exploring Splunk*.

While the process of collecting log files and syslog data are supported out-of-the-box by Splunk, there is still a need to collect, transform, and load data into Splunk that comes from a variety of other sources, in order to reduce the need to use regular expressions when analyzing data. There is also a need to transform and extract data out of Splunk into other databases and filesystems. To address these needs, Spring Integration inbound and outbound channel adapters were created, and Spring Batch support is on the road map. At the time of this writing, the Spring Integration channel adapters for Splunk are located in the GitHub repository for Spring Integration extensions. The adapters support all the ways you can get data in and out of Splunk. The

inbound adapter supports block and nonblocking searches, saved and real time searches, and the export mode. The outbound adapters support putting data into Splunk through its RESTful API, streams, or TCP. All of the functionality of Splunk is exposed via a comprehensive REST API, and several language SDKs are available to make developing with the REST API as simple as possible. The Spring Integration adapters make use of the Splunk Java SDK , also available on Github.

As an introduction to using Splunk with Spring Integration, we will create an application that stores the results from a Twitter search into Splunk. The configuration for this application is shown in Example 13-50.

Example 13-50. Configuring an application to store Twitter search results in Splunk

```
<context:property-placeholder location="twitter.properties,splunk.properties" />

<bean id="twitterTemplate"
  class="org.springframework.social.twitter.api.impl.TwitterTemplate">
  <constructor-arg value="${twitter.oauth.consumerKey}" />
  <constructor-arg value="${twitter.oauth.consumerSecret}" />
  <constructor-arg value="${twitter.oauth.accessToken}" />
  <constructor-arg value="${twitter.oauth.accessTokenSecret}" />
</bean>

<int-splunk:server id="splunkServer"
                   host="${splunk.host}" port="${splunk.port}"
                   userName="${splunk.userName}" password="${splunk.password}"
                   owner="${splunk.owner}"/>

<int:channel id="input"/>

<int:channel id="output"/>

<int-twitter:search-inbound-channel-adapter id="searchAdapter" channel="input"
             query="#voteobama OR #voteromney OR #votebieber">
  <int:poller fixed-rate="5000" max-messages-per-poll="50" />
</int-twitter:search-inbound-channel-adapter>

<int:chain input-channel="input" output-channel="output">
  <int:filter ref="tweetFilter"/>
  <int:transformer ref="splunkTransformer"/>
</int:chain>

<int-splunk:outbound-channel-adapter id="splunkOutboundChannelAdapter"
            channel="output" auto-startup="true"
            splunk-server-ref="splunkServer" pool-server-connection="true"
            sourceType="twitter-integration" source="twitter" ingest="SUBMIT"/>

<!-- tweetFilter and splunkTransformer bean definitions omitted -->
```

The Spring Social project provides the foundation for connecting to Twitter with OAuth, and also provides support for working with Facebook, LinkedIn, TripIt, Foursquare, and dozens of other social Software-as-a-Service providers. Spring Social's `TwitterTemplate` class is used by the inbound channel adapter to interact with Twitter.

This requires you to create a new application on the Twitter Developer website, in order to access the full range of functionality offered by Twitter. To connect to the Splunk server, the XML namepace `<int-splunk:server/>` is used, providing host/port information as well as user credentials (which are externally parameterized using Spring's property placeholder).

The Twitter inbound channel adapter provides support for receiving tweets as Timeline Updates, Direct Messages, Mention Messages, or Search Results. Note that there will soon be support for consuming data from the Twitter garden hose, which provides a randomly selected stream of data capped at a small percent of the full stream. In Example 13-50, the query looks for hashtags related to the United States 2012 presidental election and one hashtag to vote for Justin Bieber to win various award shows. The processing chain applies a filter and then converts the Tweet payload object into a data structure with a format optimized to help Splunk index and search the tweets. In the sample application, the filter is set to be a pass-through. The outbound channel adapter writes to the Splunk source with the REST API, specified by the attribute `inject="SUBMIT"`. The data is written to the Splunk source named *twitter* and uses the default index. You can also set the `index` attribute to specify that data should be written to the non-default index.

To run the example and see who is the most popular candidate (or pseudo-candidate), follow the directions in the directory *splunk/tweets*. Then via the Splunk web application you can create a search, such as `source="twitter" | regex tags="^voteobama$|^votebieber$|^voteromney$" | top tags`, to get a graph that shows the relative popularity of those individual hashtags.

PART VI

Data Grids

CHAPTER 14

GemFire: A Distributed Data Grid

vFabric™GemFire® (GemFire) is a commercially licensed data management platform that provides access to data throughout widely distributed architectures. It is available as a standalone product and as a component of the VMware vFabric Suite. This chapter provides an overview of Spring Data GemFire. We'll begin by introducing GemFire and some basic concepts that are prerequisite to developing with GemFire. Feel free to skip to the section "Configuring GemFire with the Spring XML Namespace" on page 258 if you are already familiar with GemFire.

GemFire in a Nutshell

GemFire provides an in-memory data grid that offers extremely high throughput, low latency data access, and scalability. Beyond a distributed cache, GemFire provides advanced features including:

- Event notification
- OQL (Object Query Language) query syntax
- Continuous queries
- Transaction support
- Remote function execution
- WAN communications
- Efficient and portable object serialization (PDX)
- Tools to aid system administrators in managing and configuring the GemFire distributed system

GemFire may be configured to support a number of distributed system topologies and is completely integrated with the Spring Framework. Figure 14-1 shows a typical client server configuration for a production LAN. The locator acts as a broker for the distributed system to support discovery of new member nodes. Client applications use the locator to acquire connections to cache servers. Additionally, server nodes use the

locator to discover each other. Once a server comes online, it communicates directly with its peers. Likewise, once a client is initialized, it communicates directly with cache servers. Since a locator is a single point of failure, two instances are required for redundancy.

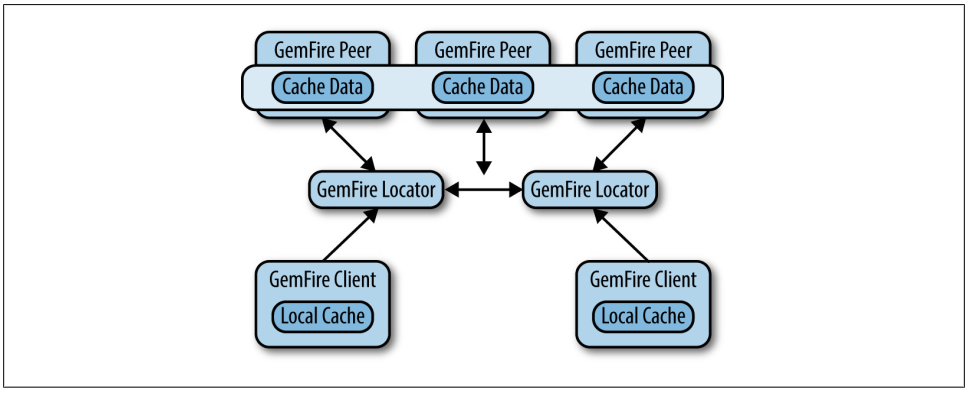


Figure 14-1. GemFire client server topology

Simple standalone configurations for GemFire are also possible. Note that the book's code samples are configured very simply as a single process with an embedded cache, suitable for development and integration testing.

In a client server scenario, the application process uses a *connection pool* (Figure 14-2) to manage connections between the client cache and the servers. The connection pool manages network connections, allocates threads, and provides a number of tuning options to balance resource usage and performance. The pool is typically configured with the address of the locator(s) [not shown in Figure 14-2]. Once the locator provides a server connection, the client communicates directly with the server. If the primary server becomes unavailable, the pool will acquire a connection to an alternate server if one is available.

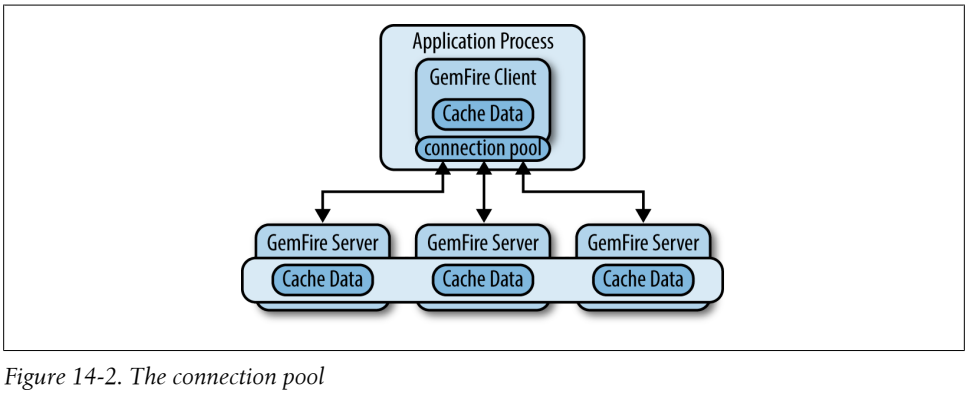


Figure 14-2. The connection pool

Caches and Regions

Conceptually, a *cache* is a singleton object that provides access to a GemFire member and offers a number of configuration options for memory tuning, network communications, and other features. The cache also acts as a container for *regions*, which provide data management and access.

A region is required to store and retrieve data from the cache. `Region` is an interface that extends `java.uti.Map` to perform basic data access using familiar key/value semantics. The `Region` interface is wired into classes that require it, so the actual region type is decoupled from the programming model (with some caveats, the discovery of which will be left as an exercise for the reader). Typically, each region is associated with one domain object, similar to a table in a relational database. Looking at the sample code, you will see three regions defined: `Customer`, `Product`, and `Order`. Note that GemFire does not manage associations or enforce relational integrity among regions.

GemFire includes the following types of regions:

Replicated
: Data is replicated across all cache members that define the region. This provides very high read performance, but writes take longer due to the need to perform the replication.

Partitioned
: Data is partitioned into buckets among cache members that define the region. This provides high read and write performance and is suitable for very large datasets that are too big for a single node.

Local
: Data exists only on the local node.

Client
: Technically, a client region is a local region that acts as a proxy to a replicated or partitioned region hosted on cache servers. It may hold data created or fetched locally; alternatively, it can be empty. Local updates are synchronized to the cache server. Also, a client region may subscribe to events in order to stay synchronized with changes originating from remote processes that access the same region.

Hopefully, this brief overview gives you a sense of GemFire's flexibility and maturity. A complete discussion of GemFire options and features is beyond the scope of this book. Interested readers will find more details on the product website.

How to Get GemFire

The vFabric GemFire website provides detailed product information, reference guides, and a link to a free developer download, limited to three node connections. For a more comprehensive evaluation, a 60-day trial version is also available.

The product download is not required to run the code samples included with this book. The GemFire jar file that includes the free developer license is available in public repositories and will be automatically downloaded by build tools such as Maven and Gradle when you declare a dependency on Spring Data GemFire. A full product install is necessary to use locators, the management tools, and so on.

Configuring GemFire with the Spring XML Namespace

Spring Data GemFire includes a dedicated XML namespace to allow full configuration of the data grid. In fact, the Spring namespace is considered the preferred way to configure GemFire, replacing GemFire's native *cache.xml* file. GemFire will continue to support *cache.xml* for legacy reasons, but you can now do everything in Spring XML and take advantage of the many wonderful things Spring has to offer, such as modular XML configuration, property placeholders, SpEL, and environment profiles. Behind the namespace, Spring Data GemFire makes extensive use of Spring's `FactoryBean` pattern to simplify the creation and initialization of GemFire components.

GemFire provides several callback interfaces, such as `CacheListener`, `CacheWriter`, and `CacheLoader` to allow developers to add custom event handlers. Using the Spring IoC container, these may configured as normal Spring beans and injected into GemFire components. This is a significant improvement over *cache.xml*, which provides relatively limited configuration options and requires callbacks to implement GemFire's `Declarable` interface.

In addition, IDEs such as the Spring Tool Suite (STS) provide excellent support for XML namespaces, such as code completion, pop-up annotations, and real-time validation, making them easy to use.

The following sections are intended to get you started using the Spring XML namespace for GemFire. For a more comprehensive discussion, please refer to the Spring Data GemFire reference guide at the project website.

Cache Configuration

To configure a GemFire cache, create a Spring bean definition file and add the Spring GemFire namespace. In STS (Figure 14-3), select the project and open the context menu (right-click) and select New→Spring Bean Configuration File. Give it a name and click Next.

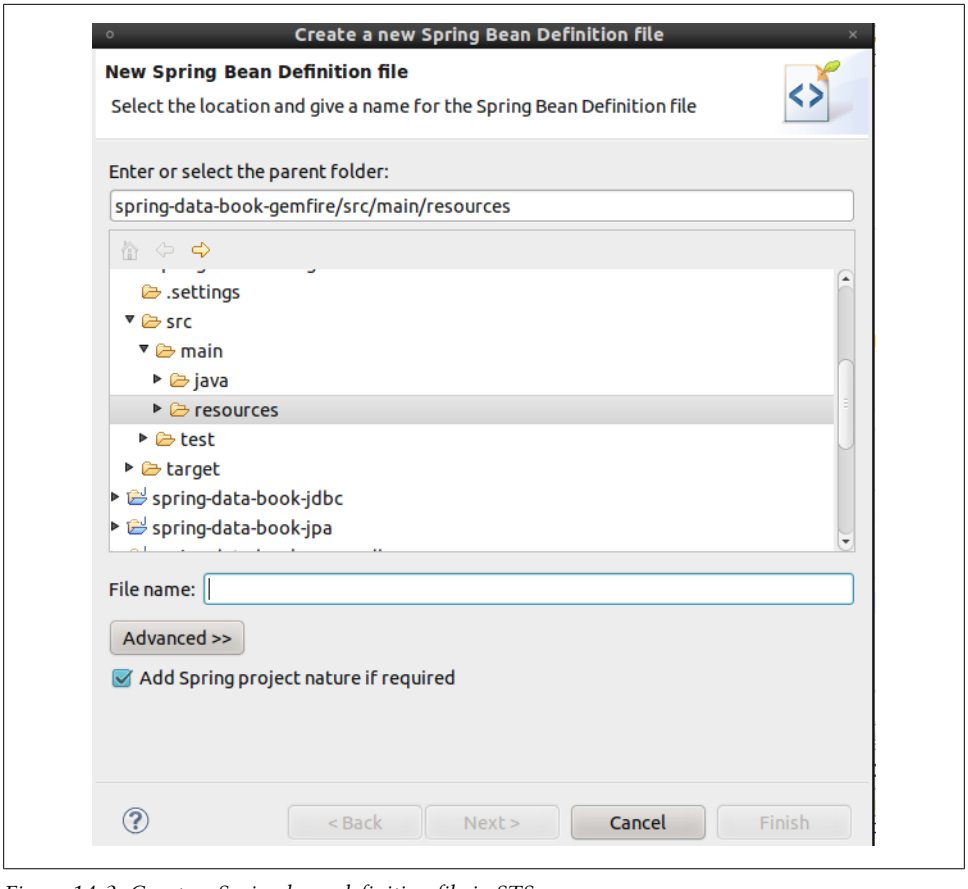


Figure 14-3. Create a Spring bean definition file in STS

In the XSD namespaces view, select the `gfe` namespace (Figure 14-4).

Notice that, in addition to the `gfe` namespace, there is a `gfe-data` namespace for Spring Data POJO mapping and repository support. The `gfe` namespace is used for core GemFire configuration.

Click Finish to open the bean definition file in an XML editor with the correct namespace declarations. (See Example 14-1.)

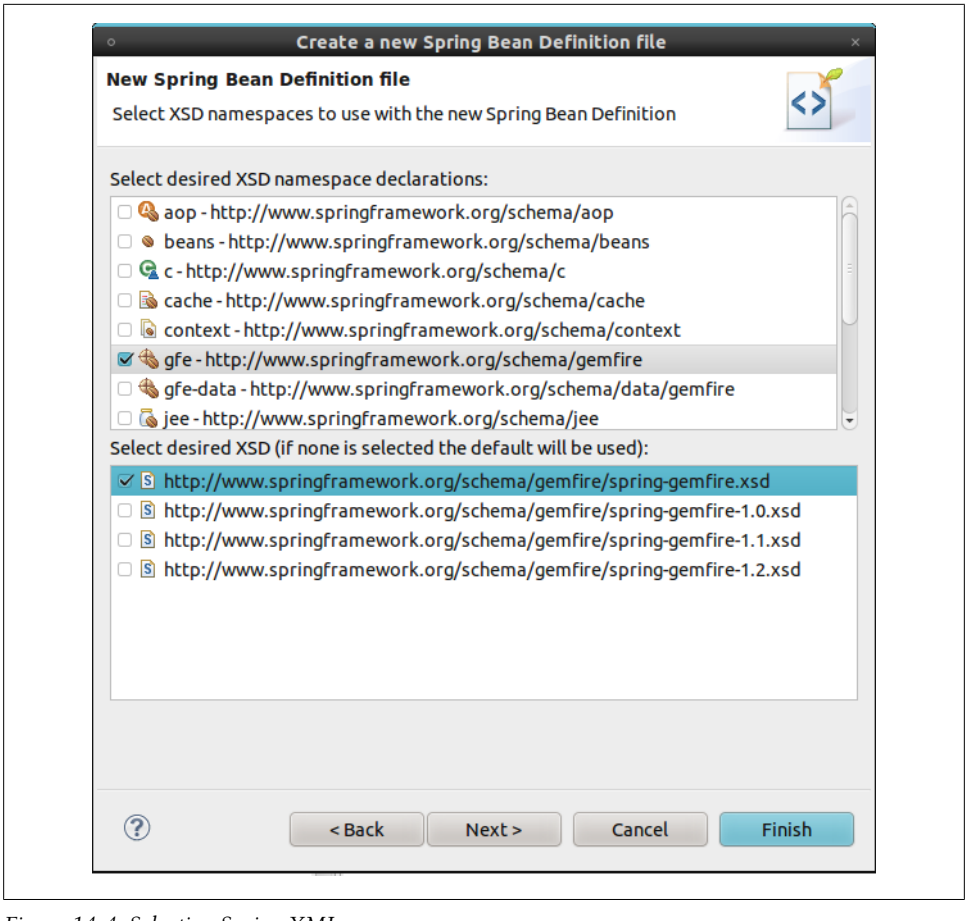


Figure 14-4. Selecting Spring XML namespaces

Example 14-1. Declaring a GemFire cache in Spring configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:gfe="http://www.springframework.org/schema/gemfire"
       xsi:schemaLocation="http://www.springframework.org/schema/gemfire
                           http://www.springframework.org/schema/gemfire/spring-gemfire.xsd
                           http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

   <gfe:cache/>

</beans>
```

Now use the `gfe` namespace to add a `cache` element. That's it! This simple cache declaration will create an embedded cache, register it in the Spring `ApplicationContext` as `gemfireCache`, and initialize it when the context is created.

Prior releases of Spring Data GemFire created default bean names using hyphens (e.g., `gemfire-cache`). As of the 1.2.0 release, these are replaced with camelCase names to enable autowiring via annotations (`@Autowired`). The old-style names are registered as aliases to provide backward compatibility.

You can easily change the bean name by setting the `id` attribute on the `cache` element. However, all other namespace elements assume the default name unless explicitly overridden using the `cache-ref` attribute. So you can save yourself some work by following the convention.

The `cache` element provides some additional attributes, which STS will happily suggest if you press Ctrl-Space. The most significant is the `properties-ref` attribute. In addition to the API, GemFire exposes a number of global configuration options via external properties. By default, GemFire looks for a file called *gemfire.properties* in all the usual places: the user's home directory, the current directory, and the classpath. While this is convenient, it may result in unintended consequences if you happen to have these files laying around. Spring alleviates this problem by offering several better alternatives via its standard property loading mechanisms. For example, you can simply construct a `java.util.Properties` object inline or load it from a properties file located on the classpath or filesystem. Example 14-2 uses properties to configure GemFire logging.

Example 14-2. Referencing properties to configure GemFire

Define properties inline:

```
<util:properties id="props">
   <prop key="log-level">info</prop>
   <prop key="log-file">gemfire.log</prop>
</util:properties>

<gfe:cache properties-ref="props" />
```

Or reference a resource location:

```
<util:properties id="props" location="gemfire-cache.properties" />

<gfe:cache properties-ref="props" />
```

It is generally preferable to maintain properties of interest to system administrators in an agreed-upon location in the filesystem rather than defining them in Spring XML or packaging them in *.jar* files.

Note the use of Spring's `util` namespace to create a `Properties` object. This is related to, but not the same as, Spring's property placeholder mechanism, which uses token-based substitution to allow properties on any bean to be defined externally from a variety of sources. Additionally, the `cache` element includes a `cache-xml-location` attribute to enable the cache to be configured with GemFire's native configuration schema. As previously noted, this is mostly there for legacy reasons.

The cache element also provides some `pdx-*` attributes required to enable and configure GemFire's proprietary serialization feature (PDX). We will address PDX in "Repository Usage" on page 271.

For advanced cache configuration, the cache element provides additional attributes for tuning memory and network communications (shown in Figure 14-5) and child elements to register callbacks such as `TransactionListers`, `and TransactionWriters` (Figure 14-6).

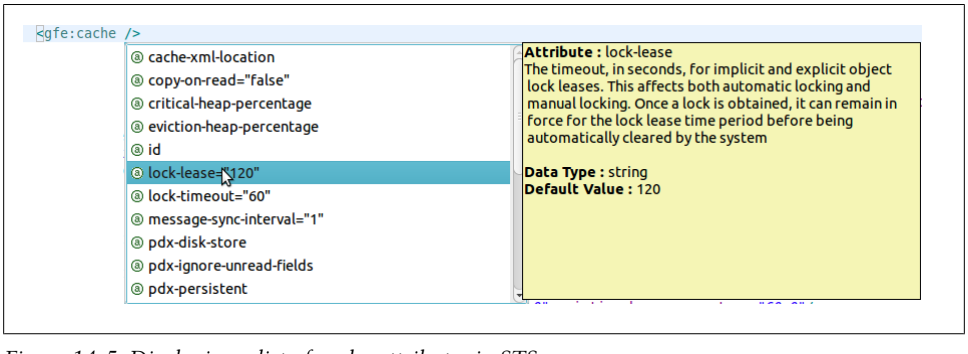

<gfe:cache />
cache-xml-location
copy-on-read="false"
critical-heap-percentage
eviction-heap-percentage
id
lock-lease="120"
lock-timeout="60"
message-sync-interval="1"
pdx-disk-store
pdx-ignore-unread-fields
pdx-persistent
Attribute : lock-lease
The timeout, in seconds, for implicit and explicit object lock leases. This affects both automatic locking and manual locking. Once a lock is obtained, it can remain in force for the lock lease time period before being automatically cleared by the system
Data Type : string
Default Value : 120

Figure 14-5. Displaying a list of cache attributes in STS

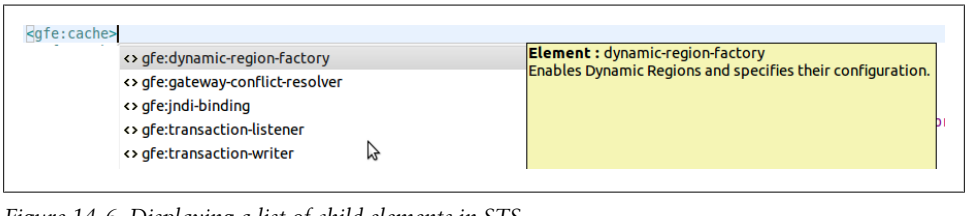


Figure 14-6. Displaying a list of child elements in STS

The `use-bean-factory-locator` attribute (not shown) deserves a mention. The factory bean responsible for creating the cache uses an internal Spring type called a `BeanFactoryLocator` to enable user classes declared in GemFire's native *cache.xml* file to be registered as Spring beans. The `BeanFactoryLocator` implementation also permits only one bean definition for a cache with a given `id`. In certain situations, such as running JUnit integration tests from within Eclipse, you'll need to disable the `BeanFactoryLocator` by setting this value to `false` to prevent an exception. This exception may also arise during JUnit tests running from a build script. In this case, the test runner should be configured to fork a new JVM for each test (in Maven, set `<forkmode>always</forkmode>`). Generally, there is no harm in setting this value to `false`.

Region Configuration

As mentioned in the chapter opener, GemFire provides a few types of regions. The XML namespace defines `replicated-region`, `partitioned-region`, `local-region`, and `client-region` elements to create regions. Again, this does not cover all available features but highlights some of the more common ones. A simple region declaration, as shown in Example 14-3, is all you need to get started.

Example 14-3. Basic region declaration

```
<gfe:cache/>
<gfe:replicated-region id="Customer" />
```

The region has a dependency on the cache. Internally, the cache creates the region. By convention, the namespace does the wiring implicitly. The default cache declaration creates a Spring bean named `gemfireCache`. The default region declaration uses the same convention. In other words, Example 14-3 is equivalent to:

```
<gfe:cache id="gemfireCache" />
<gfe:replicated-region id="Customer" cache-ref="gemfireCache" />
```

If you prefer, you can supply any valid bean name, but be sure to set `cache-ref` to the corresponding bean name as required.

Typically, GemFire is deployed as a distributed data grid, hosting replicated or partitioned regions on cache servers. Client applications use client regions to access data. For development and integration testing, it is a best practice to eliminate any dependencies on an external runtime environment. You can do this by simply declaring a replicated or local region with an embedded cache, as is done in the sample code. Spring environment profiles are extremely useful in configuring GemFire for different environments.

In Example 14-4, the `dev` profile is intended for integration testing, and the `prod` profile is used for the deployed cache configuration. The cache and region configuration is transparent to the application code. Also note the use of property placeholders to

specify the locator hosts and ports from an external properties file. Cache client configuration is discussed further in "Cache Client Configuration" on page 265.

Example 14-4. Sample XML configuration for development and production

```
<beans profile="dev">
  <gfe:cache/>
  <gfe:replicated-region id="Customer" />
</beans>

<beans profile="prod">
  <context:properties-placeholder location="client-app.properties" />
  <gfe:client-cache pool-name="pool" />

  <gfe:client-region id="Customer" />

  <gfe:pool id="pool">
    <gfe:locator host="${locator.host.1}" port="${locator.port.1}"/>
    <gfe:locator host="${locator.host.2}" port="${locator.port.2}"/>
 </gfe:pool>
</beans>
```

Spring provides a few ways to activate the appropriate environment profile(s). You can set the property `spring.profiles.active` in a system property, a servlet context parameter, or via the `@ActiveProfiles` annotation.

As shown in Figure 14-7, there are a number of common region configuration options as well as specific options for each type of region. For example, you can configure all regions to back up data to a local disk store synchronously or asynchronously.

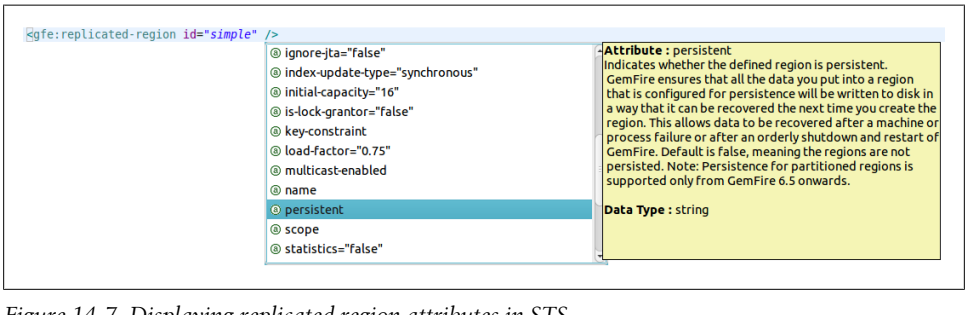

<gfe:replicated-region id="simple" />
ignore-jta="false"
index-update-type="synchronous"
initial-capacity="16"
is-lock-grantor="false"
key-constraint
load-factor="0.75"
multicast-enabled
name
persistent
scope
statistics="false"
Attribute : persistent
Indicates whether the defined region is persistent. GemFire ensures that all the data you put into a region that is configured for persistence will be written to disk in a way that it can be recovered the next time you create the region. This allows data to be recovered after a machine or process failure or after an orderly shutdown and restart of GemFire. Default is false, meaning the regions are not persisted. Note: Persistence for partitioned regions is supported only from GemFire 6.5 onwards.
Data Type : string

Figure 14-7. Displaying replicated region attributes in STS

Additionally, you may configure regions to synchronize selected entries over a WAN gateway to distribute data over a wide geographic area. You may also register `CacheListeners`, `CacheLoaders`, and `CacheWriters` to handle region events. Each of these interfaces is used to implement a callback that gets invoked accordingly. A `CacheListener` is a generic event handler invoked whenever an entry is created, updated,

destroyed, etc. For example, you can write a simple `CacheListener` to log cache events, which is particularly useful in a distributed environment (see Example 14-5). A `CacheLoader` is invoked whenever there is a cache miss (i.e., the requested entry does not exist), allowing you to "read through" to a database or other system resource. A `CacheWriter` is invoked whenever an entry is updated or created to provide "write through" or "write behind" capabilities.

Example 14-5. LoggingCacheListener implementation

```
public class LoggingCacheListener extends CacheListenerAdapter {

  private static Log log = LogFactory.getLog(LoggingCacheListener.class);

  @Override
  public void afterCreate(EntryEvent event) {
     String regionName = event.getRegion().getName();
     Object key = event.getKey();
     Object newValue = event.getNewValue();
     log.info("In region [" + regionName + "] created key ["
       + key + "] value [" + newValue + "]");
   }

   @Override
   public void afterDestroy(EntryEvent event) {
    …
   }

   @Override
   public void afterUpdate(EntryEvent event) {
    …
   }
}
```

Other options include *expiration*, the maximum time a region or an entry is held in the cache, and *eviction*, policies that determine which items are removed from the cache when the defined memory limit or the maximum number of entries is reached. Evicted entries may optionally be stored in a disk overflow.

You can configure partitioned regions to limit the amount of local memory allocated to each partition node, define the number of buckets used, and more. You may even implement your own `PartitionResolver` to control how data is colocated in partition nodes.

Cache Client Configuration

In a client server configuration, application processes are *cache clients*—that is, they produce and consume data but do not distribute it directly to other processes. Neither does a cache client implicitly see updates performed by remote processes. As you might expect by now, this is entirely configurable. Example 14-6 shows a basic client-side setup using a `client-cache`, `client-region`, and a `pool`. The `client-cache` is a

lightweight implementation optimized for client-side services, such as managing one or more client regions. The `pool` represents a connection pool acting as a bridge to the distributed system and is configured with any number of locators.

Typically, two locators are sufficient: the first locator is primary, and the remaining ones are strictly for failover. Every distributed system member should use the same locator configuration. A locator is a separate process, running in a dedicated JVM, but is not strictly required. For development and testing, the pool also provides a `server` child element to access cache servers directly. This is useful for setting up a simple client/server environment (e.g., on your local machine) but not recommended for production systems. As mentioned in the chapter opener, using a locator requires a full GemFire installation, whereas you can connect to a server directly just using the APIs provided in the publicly available *gemfire.jar* for development, which supports up to three cache members.

Example 14-6. Configuring a cache pool

```
<gfe:client-cache pool-name="pool" />

<gfe:client-region id="Customer" />

<gfe:pool id="pool">
  <gfe:locator host="${locator.host.1}" port="${locator.port.1}"/>
  <gfe:locator host="${locator.host.2}" port="${locator.port.2}"/>
</gfe:pool>
```

You can configure the `pool` to control thread allocation for connections and network communications. Of note is the `subscription-enabled` attribute, which you must set to `true` to enable synchronizing region entry events originating from remote processes (Example 14-7).

Example 14-7. Enabling subscriptions on a cache pool

```
<gfe:client-region id="Customer">
   <gfe:key-interest durable="false" receive-values="true" />
</client-region>

<gfe:pool id="pool" subcription-enabled="true">
  <gfe:locator host="${locator.host.1}" port="${locator.port.1}"/>
  <gfe:locator host="${locator.host.2}" port="${locator.port.2}"/>
</gfe:pool>
```

With subscriptions enabled, the `client-region` may register interest in all keys or specific keys. The subscription may be durable, meaning that the `client-region` is updated with any events that may have occurred while the client was offline. Also, it is possible to improve performance in some cases by suppressing transmission of values unless

explicitly retrieved. In this case, new keys are visible, but the value must be retrieved explicitly with a `region.get(key)` call, for example.

Cache Server Configuration

Spring also allows you to create and initialize a cache server process simply by declaring the cache and region(s) along with an additional `cache-server` element to address server-side configuration. To start a cache server, simply configure it using the namespace and start the application context, as shown in Example 14-8.

Example 14-8. Bootstrapping a Spring application context

```
public static void main(String args[]) {
   new ClassPathXmlApplicationContext("cache-config.xml");
}
```

Figure 14-8 shows a Spring-configured cache server hosting two partitioned regions and one replicated region. The `cache-server` exposes many parameters to tune network communications, system resources, and the like.

Figure 14-8. Configuring a cache server

WAN Configuration

WAN configuration is required for geographically distributed systems. For example, a global organization may need to share data across the London, Tokyo, and New York offices. Each location manages its transactions locally, but remote locations need to be synchronized. Since WAN communications can be very costly in terms of performance and reliability, GemFire queues events, processed by a WAN gateway to achieve

eventual consistency. It is possible to control which events get synchronized to each remote location. It is also possible to tune the internal queue sizes, synchronization scheduling, persistent backup, and more. While a detailed discussion of GemFire's WAN gateway architecture is beyond the scope of this book, it is important to note that WAN synchronization must be enabled at the region level. See Example 14-9 for a sample configuration.

Example 14-9. GemFire WAN configuration

```
<gfe:replicated-region id="region-with-gateway" enable-gateway="true" hub-id="gateway-hub" />

<gfe:gateway-hub id="gateway-hub" manual-start="true">
  <gfe:gateway gateway-id="gateway">
    <gfe:gateway-listener>
       <bean class="..."/>
    </gfe:gateway-listener>
    <gfe:gateway-queue maximum-queue-memory="5" batch-size="3" batch-time-interval="10" />
  </gfe:gateway>

  <gfe:gateway gateway-id="gateway2">
    <gfe:gateway-endpoint port="1234" host="host1" endpoint-id="endpoint1" />
    <gfe:gateway-endpoint port="2345" host="host2" endpoint-id="endpoint2" />
  </gfe:gateway>
</gfe:gateway-hub>
```

This example shows a region enabled for WAN communications using the APIs available in GemFire 6 versions. The `enable-gateway` attribute must be set to `true` (or is implied by the presence of the `hub-id` attribute), and the `hub-id` must reference a `gateway-hub` element. Here we see the `gateway-hub` configured with two gateways. The first has an optional `GatewayListener` to handle gateway events and configures the gateway queue. The second defines two remote gateway endpoints.

The WAN architecture will be revamped in the upcoming GemFire 7.0 release. This will include new features and APIs, and will generally change the way gateways are configured. Spring Data GemFire is planning a concurrent release that will support all new features introduced in GemFire 7.0. The current WAN architecture will be deprecated.

Disk Store Configuration

GemFire allows you to configure disk stores for persistent backup of regions, disk overflow for evicted cache entries, WAN gateways, and more. Because a disk store may serve multiple purposes, it is defined as a top-level element in the namespace and may be referenced by components that use it. Disk writes may be synchronous or asynchronous.

For asynchronous writes, entries are held in a queue, which is also configurable. Other options control scheduling (e.g., the maximum time that can elapse before a disk write

is performed or the maximum file size in megabytes). In Example 14-10, an overflow disk store is configured to store evicted entries. For asynchronous writes, it will store up to 50 entries in a queue, which will be flushed every 10 seconds or if the queue is at capacity. The region is configured for eviction to occur if the total memory size exceeds 2 GB. A custom `ObjectSizer` is used to estimate memory allocated per entry.

Example 14-10. Disk store configuration

```
<gfe:partitioned-region id="partition-data" persistent="true" disk-store-ref="ds2">
  <gfe:eviction type="MEMORY_SIZE" threshold="2048" action="LOCAL_DESTROY">
  <gfe:object-sizer>
    <bean class="org.springframework.data.gemfire.SimpleObjectSizer" />
  </gfe:object-sizer>
  </gfe:eviction>
</gfe:partitioned-region>

<gfe:disk-store id="ds2" queue-size="50" auto-compact="true"
     max-oplog-size="10" time-interval="10000">
  <gfe:disk-dir location="/gemfire/diskstore" />
</gfe:disk-store>
```

Data Access with GemfireTemplate

Spring Data GemFire provides a template class for data access, similar to the `JdbcTemplate` or `JmsTemplate`. The `GemfireTemplate` wraps a single region and provides simple data access and query methods as well as a callback interface to access region operations. One of the key reasons to use the `GemfireTemplate` is that it performs exception translation from GemFire checked exceptions to Spring's `PersistenceException` runtime exception hierarchy. This simplifies exception handling required by the native `Region` API and allows the template to work more seamlessly with the Spring declarative transactions using the `GemfireTransactionManager` which, like all Spring transaction managers, performs a rollback for runtime exceptions (but not checked exceptions) by default. Exception translation is also possible for `@Repository` components, and transactions will work with `@Transactional` methods that use the `Region` interface directly, but it will require a little more care.

Example 14-11 is a simple demonstration of a data access object wired with the `GemfireTemplate`. Notice the `template.query()` invocation backing `findByLastName(...)`. Queries in GemFire use the Object Query Language (OQL). This method requires only a boolean predicate defining the query criteria. The body of the query, `SELECT * from [region name] WHERE...`, is assumed. The template also implements `find(...)` and `findUnique(...)` methods, which accept parameterized query strings and associated parameters and hide GemFire's underlying `QueryService` API.

Example 14-11. Repository implementation using GemfireTemplate

```
@Repository
class GemfireCustomerRepository implements CustomerRepository {
```

```
  private final GemfireTemplate template;

  @Autowired
  public GemfireCustomerRepository(GemfireTemplate template) {
    Assert.notNull(template);
    this.template = template;
  }

  /**
   * Returns all objects in the region. Not advisable for very large datasets.
   */
  public List<Customer> findAll() {
    return new ArrayList<Customer>((Collection<? extends Customer>) ↪
template.getRegion().values());
  }

  public Customer save(Customer customer) {
    template.put(customer.getId(), customer);
    return customer;
  }

  public List<Customer> findByLastname(String lastname) {

    String queryString = "lastname = '" + lastname + "'";
    SelectResults<Customer> results = template.query(queryString);
    return results.asList();
  }

  public Customer findByEmailAddress(EmailAddress emailAddress) {

    String queryString = "emailAddress = ?1";
    return template.findUnique(queryString, emailAddress);
  }

  public void delete(Customer customer) {
    template.remove(customer.getId());
  }
}
```

We can configure the `GemfireTemplate` as a normal Spring bean, as shown in Example 14-12.

Example 14-12. GemfireTemplate configuration

```
<bean id="template" class="org.springframework.data.gemfire.GemfireTemplate">
  <property name="region" ref="Customer" />
</bean>
```

Repository Usage

The 1.2.0 release of Spring Data GemFire introduces basic support for Spring Data repositories backed by GemFire. All the core repository features described in Chapter 2 are supported, with the exception of paging and sorting. The sample code demonstrates these features.

POJO Mapping

Since GemFire regions require a unique key for each object, the top-level domain objects—`Customer`, `Order`, and `Product`—all inherit from `AbstractPersistentEntity`, which defines an `id` property (Example 14-13).

Example 14-13. AbstractPersistentEntity domain class

```
import org.springframework.data.annotation.Id;

public class AbstractPersistentEntity {

  @Id
  private final Long id;
}
```

Each domain object is annotated with `@Region`. By convention, the region name is the same as the simple class name; however, we can override this by setting the annotation value to the desired region name. This must correspond to the region name—that is, the value of `id` attribute or the `name` attribute, if provided, of the region element. Common attributes, such as `@PersistenceConstructor` (shown in Example 14-14) and `@Transient`, work as expected.

Example 14-14. Product domain class

```
@Region
public class Product extends AbstractPersistentEntity {

  private String name, description;
  private BigDecimal price;
  private Map<String, String> attributes = new HashMap<String, String>();

  @PersistenceConstructor
  public Product(Long id, String name, BigDecimal price, String description) {

    super(id);
    Assert.hasText(name, "Name must not be null or empty!");
    Assert.isTrue(BigDecimal.ZERO.compareTo(price) < 0, "Price must be greater than zero!");

    this.name = name;
    this.price = price;
    this.description = description;
}
```

Creating a Repository

GemFire repositories support basic CRUD and query operations, which we define using Spring Data's common method name query mapping mechanism. In addition, we can configure a repository method to execute any OQL query using `@Query`, as shown in Example 14-15.

Example 14-15. ProductRepository interface

```
public interface ProductRepository extends CrudRepository<Product, Long> {

  List<Product> findByDescriptionContaining(String description);

  /**
   * Returns all {@link Product}s having the given attribute value.
   * @param attribute
   * @param value
   * @return
   */
  @Query("SELECT * FROM /Product where attributes[$1] = $2")
  List<Product> findByAttributes(String key, String value);

  List<Product> findByName(String name);
}
```

You can enable repository discovery using a dedicated `gfe-data` namespace, which is separate from the core `gfe` namespace. Alternatively, if you're using Java configuration, simply annotate your configuration class with `@EnableGemfireRepositories`, as shown in Example 14-16.

Example 14-16. Enabling GemFire repositories using XML

```
<gfe-data:repositories base-package="com.oreilly.springdata.gemfire" />
```

PDX Serialization

PDX is GemFire's proprietary serialization library. It is highly efficient, configurable, interoperable with GemFire client applications written in C# or C++, and supports object versioning. In general, objects must be serialized for operations requiring network transport and disk persistence. Cache entries, if already serialized, are stored in serialized form. This is generally true with a distributed topology. A standalone cache with no persistent backup generally does not perform serialization.

If PDX is not enabled, Java serialization will be used. In this case, your domain classes and all nontransient properties must implement `java.io.Serializable`. This is not a requirement for PDX. Additionally, PDX is highly configurable and may be customized to optimize or enhance serialization to satisfy your application requirements.

Example 14-17 shows how to set up a GemFire repository to use PDX.

Example 14-17. Configuring a MappingPdxSerializer

```
<gfe:cache pdx-serializer="mapping-pdx-serializer" />

<bean id="mapping-pdx-serializer"
      class="org.springframework.data.gemfire.mapping.MappingPdxSerializer" />
```

The `MappingPdxSerializer` is automatically wired with the default mapping context used by the repositories. One limitation to note is that each cache instance can have only one PDX serializer, so if you're using PDX for repositories, it is advisable to set up a dedicated cache node (i.e., don't use the same process to host nonrepository regions).

Continuous Query Support

A very powerful feature of GemFire is its support for continuous queries (CQ), which provides a query-driven event notification capability. In traditional distributed applications, data consumers that depend on updates made by other processes in near-real time have to implement some type of polling scheme. This is not particularly efficient or scalable. Alternatively, using a publish-subscribe messaging system, the application, upon receiving an event, typically has to access related data stored in a disk-based data store. Continuous queries provide an extremely efficient alternative. Using CQ, the client application registers a query that is executed periodically on cache servers. The client also provides a callback that gets invoked whenever a region event affects the state of the query's result set. Note that CQ requires a client/server configuration.

Spring Data GemFire provides a `ContinuousQueryListenerContainer`, which supports a programming model based on Spring's `DefaultMessageListenerContainer` for JMS-message-driven POJOs. To configure CQ, create a CQLC using the namespace, and register a listener for each continuous query (Example 14-18). Notice that the pool must have `subscription-enabled` set to `true`, as CQ uses GemFire's subscription mechanism.

Example 14-18. Configuring a ContinuousQueryListenerContainer

```
<gfe:client-cache pool-name="client-pool" />

<gfe:pool id="client-pool" subscription-enabled="true">
  <gfe:server host="localhost" port="40404" />
</gfe:pool>

<gfe:client-region id="Person" pool-name="client-pool" />

<gfe:cq-listener-container>
  <gfe:listener ref="cqListener" query="select * from /Person" />
</gfe:cq-listener-container>

<bean id="cqListener" class="org.springframework.data.gemfire.examples.CQListener" />
```

Now, implement the listener as shown in Example 14-19.

Example 14-19. Continuous query listener implementation

```
public class CQListener {

  private static Log log = LogFactory.getLog(CQListener.class);

  public void handleEvent(CqEvent event) {
     log.info("Received event " + event);
  }
}
```

The `handleEvent()` method will be invoked whenever any process makes a change in the range of the query. Notice that `CQListener` does not need to implement any interface, nor is there anything special about the method name. The continuous query container is smart enough to automatically invoke a method that has a single `CqEvent` parameter. If there is more than one, declare the method name in the `listener` configuration.

Bibliography

[ChoDir10] Chodorow, Kristina, and Michael Dirolf. *MongoDB: The Definitive Guide*, O'Reilly Media, 2010. *http://shop.oreilly.com/product/0636920001096.do*

[CoTeGreBa11] Cogoluegnes, Arnaud, Thierry Templier, Gary Gregory, and Olivier Bazoud. *Spring Batch in Action*, Manning Publications Co., 2011. *http://www.manning.com/templier*

[Evans03] Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003.

[Fielding00] Fielding, Roy. *Architectural Styles and the Design of Network-Based Software Architectures*, University of California, 2000. *http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm*

[Fisher12] Fisher, Mark. *Spring Integration in Action*, Manning Publications Co., 2012. *http://www.manning.com/fisher*

[Hunger12] Hunger, Michael. *Good Relationships: The Spring Data Neo4j Guide Book*, InfoQ, 2012. *http://www.infoq.com/minibooks/good-relationships-spring-data*

[Konda12] Konda, Madhusudhan. *Just Spring Data Access*, O'Reilly Media, 2012. *http://shop.oreilly.com/product/0636920025405.do*

[LongMay11] Long, Josh, and Steve Mayzak. *Getting Started with Roo*, O'Reilly Media, 2011. *http://shop.oreilly.com/product/0636920020981.do*

[Minella11] Minella, Michael T.. *Pro Spring Batch*, Apress Media LLC, 2011. *http://www.apress.com/9781430234524*

[RimPen12] Rimple, Ken, and Srini Penchikala. *Spring Roo in Action*, Manning Publications Co., 2012. *http://www.manning.com/rimple*

[WePaRo10] Webber, Jim, Savas Parastatidis, and Ian Robinson. *REST in Practice*, O'Reilly Media, 2010. *http://shop.oreilly.com/product/9780596805838.do*

Index

A

AbstractDocument class, 83
AbstractEntity class
  equals method, 38, 106
  hashCode method, 38, 106
AbstractIntegrationTest class, 48
AbstractMongoConfiguration class, 81, 82
AbstractPersistentEntity class, 271
@ActiveProfile annotation, 42
@ActiveProfiles annotation, 264
aggregate root, 40
Amazon Elastic Map Reduce, 176
analyzing data (see data analysis)
Annotation Processing Toolkit (APT)
  about, 30
  supported annotation processors, 31, 99
AnnotationConfigApplicationContext class, 46
AnnotationConfigWebApplicationContext class, 160
annotations (see specific annotations)
Apache Hadoop project (see Hadoop; Spring for Apache Hadoop)
Appassembler plug-in, 182
application-context.xml file, 95
ApplicationConfig class, 46, 47, 95, 160
ApplicationContext interface
  cache declaration, 261
  caching interceptor and, 137
  derived finder methods and, 118
  launching MapReduce jobs, 184
  mongoDBFactory bean, 90
  registering converters in, 108
  REST repository exporter and, 160
ApplicationConversionServiceFactoryBean.java, 150
APT (Annotation Processing Toolkit)
  about, 30
  supported annotation processors, 31, 99
AspectJ extension (Java)
  Spring Data Neo4j and, 105, 123
  Spring Roo and, 141, 149
@Async annotation, 209
Atom Syndication Format RFC, 161
atomic counters, 134
AtomicInteger class, 134
AtomicLong class, 134
Autosys job scheduler, 191
@Autowired annotation, 65
awk command, 177, 204

B

BASE acronym, xvii
bash shell, 205
BasicDBObject class, 80, 89
BeanFactoryLocator interface, 262
BeanPropertyItemSqlParameterSourceProvider class, 248
BeanPropertyRowMapper class, 57, 70
BeanWrapperFieldExtractor class, 237
Bergh-Johnsson, Dan, 84
Big Data
  about, xvi
  analyzing data with Hadoop, 195–217
  integrating Hadoop with Spring Batch and Spring Integration, 219–250
  Spring for Apache Hadoop, 175–193
BigInteger class, 83
BigTable technology (Google), 214

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

BooleanExpression class, 57
BoundHashOperations interface, 130
BoundListOperations interface, 130
BoundSetOperations interface, 130
BoundValueOperations interface, 130
BoundZSetOperations interface, 130
BSON (binary JSON), 77
build system
  generating query metamodel, 30
  generating query types, 58

C

@Cacheable annotation, 136
CacheListener interface, 258, 264
CacheLoader interface, 258, 264
caches
  about, 257
  configuring for GemFire, 258–262
  Redis and, 136
  regions and, 257
CacheWriter interface, 258, 264
ClassLoader class, 134
collaborative filtering, 121
Collection interface, 116
CollectionCallback interface, 95
@Column annotation, 38, 40
command line, Spring Roo, 143
CommandLineJobRunner class, 242
@ComponentScan annotation, 45
@CompoundIndex annotation, 91
@Configuration annotation, 45, 137
Configuration class, 180, 183, 197
<configuration /> element, 184
CONNECT verb (HTTP), 157
ConnectionFactory interface, 129
Content-Type header (HTTP), 163, 171
<context:component-scan /> element, 198
@ComponentScan annotation, 47
ContextLoaderListener class, 160
ContinuousQueryListenerContainer class, 273
Control-M job scheduler, 191
copyFromLocal command, 178
counter, atomic, 134
CqEvent interface, 274
CQListener interface, 274
cron utility, 191
CronTrigger class, 191
CRUD operations
  about, 13
  exposing, 19, 48
  REST repository exporter and, 157
  Spring Data Neo4j support, 113
  transactions and, 50
CrudRepository interface
  about, 19
  delete... methods, 19
  executing predicates, 33
  extending, 97
  findAll method, 19
  findOne method, 48
  REST repository exporter and, 160
  save method, 19, 48
curl command, 161, 231
Cypher query language
  derived and annotated finder methods, 116–119
  Neo4j support, 103
  query components, 114

D

DAO (Data Access Exception) exception hierarchy, 209
data access layers
  CRUD operations, 13
  defining query methods, 16–18
  defining repositories, 18–22
  IDE integration, 22–24
  implementing, 13–15
  Spring Roo and, 143
data analysis
  about, 195
  using HBase, 214–217
  using Hive, 195–204
  using Pig, 204–213
data cleansing, 176
data grids (see Spring Data GemFire)
data stores
  graph databases as, 101
  querying using Querydsl, 32
DataAccessException class, 37, 43, 65, 94
database schemas, 59, 101
DataSource interface, 201, 235
DB class, 80, 95
<db-factory /> element, 82
DbCallback class, 95
DBCollection class, 80, 95
DBObject interface
  creating MongoDB document, 80

custom conversion, 95
customizing conversion, 91
mapping subsystem and, 83
@DBRef annotation, 88, 89
Declarable interface, 258
DefaultMessageListenerContainer class, 273
Delete class, 215
DELETE verb (HTTP), 157, 160, 171
deleting objects, 73
Dev HTTP Client for Google Chrome plug-in, 160
Dialect class (Hibernate), 57
DirectChannel class, 212
disk stores configuration, 268
DistCp class, 189
@Document annotation
APT processors and, 31
mapping subsystem example, 85, 88, 91
DocumentCallbackHandler interface, 95
Domain Driven Design (Evans), 39
domain model (Spring Data project), 6
Driver interface, 201
DUMP operation, 206
DuplicateKeyException class, 91
DynamicProperties interface, 108

E

EclipseLink project, 37
Eifrem, Emil, 3, 105
Elastic Map Reduce (Amazon), 176
@ElementCollection annotation, 49
@Embeddable annotation, 31, 39, 150
EmbeddedDatabaseBuilder class, 54
@EmbeddedOnly annotation, 31
@EnableGemfireRepositories annotation, 272
@EnableJpaRepositories annotation, 47
@EnableMongoRepositories annotation, 96
@EnableTransactionManagement annotation, 45, 47
@Enable...Repositories annotation, 14
Enterprise Integration Patterns, 220
@Entity annotation, 28, 31, 38
entity lifecycle
about, 122
creating entities, 148–150, 153
EntityManager interface
JPA repositories and, 37, 43
querying relational stores, 32
EntityManagerFactory interface, 37, 44, 45
ENV environment variable, 186
Evans, Eric, 39
event forwarding, 229
event streams, 226–229
ExampleDriver class, 179, 184
executing queries (see query execution)
ExecutorChannel class, 212
ExitStatus codes, 240
exporting data from HDFS, 243–250
Expression interface, 57

F

Facebook networking site, 195
@Fetch annotation
about, 110
advanced mapping mode and, 124
fetch strategies and, 123
@Field annotation, 85, 86
Fielding, Roy, 157
FileSystem class, 187, 236
filtering, collaborative, 121
finder methods
annotated, 116
derived, 117–119
result handling, 116
FlatFileItemReader class, 233, 244, 245
FlatFileParseException class, 245
FsShell class, 187, 222
FsShellWritingMessageHandler class, 223

G

GemFire data grid, 255–257
GemfireTemplate class, 269
GemfireTransactionManager class, 269
Get class, 215
GET commands (Redis), 129
get product-counts command, 132
GET verb (HTTP), 157
getmerge command, 178
Getting Started with Roo (Long and Mayzak), 141
gfe namespace, 259
gfe-data namespace, 259
Google BigTable technology, 214
Google File System, 175
graph databases, 101
(see also Neo4j graph database)
GraphDatabaseService interface, 111, 112

@GraphId annotation, 106, 122
@GraphProperty annotation, 108
GraphRepository interface, 106, 113
GROUP operation, 206
Grunt console (Pig), 204, 206

H

Hadoop
  about, 175
  additional resources, 177
  analyzing data with, 195–217
  challenges developing with, 176
  collecting and loading data into HDFS, 219–237
  exporting data from HDFS, 243–250
  Spring Batch support, 238–240
  wordcount application, 175, 177–183, 240–242
  workflow considerations, 238–243
hadoop command, 178
Hadoop command-line, 183
Hadoop Distributed File System (see HDFS)
Hadoop MapReduce API
  about, 175
  data analysis and, 195
  Hive and, 195
  Pig and, 204
  Spring Batch and, 240–242
  wordcount application and, 175, 177–183
HAProxy load balancer, 197
HashOperations interface, 130, 133
HATEOAS acronym, 158
HBase database
  about, 195, 214
  installing, 214
  Java client, 215–217
hbase shell command, 214
HBaseConfiguration class, 216
HBaseTemplate class, 216
HDFS (Hadoop Distributed File System)
  about, 175
  challenges developing with, 176
  collecting and loading data into, 219–237
  combining scripting and job submission, 190
  event forwarding and, 229
  event streams and, 226–229
  exporting data from, 243–250
  file storage and, 177
  Hive project and, 195
  loading data from relational databases, 234–237
  loading log files into, 222–226
  Pig project and, 206
  scripting on JVM, 187–190
  Spring Batch and, 234, 240–242
  Spring Integration and, 222
HDFS shell, 177, 182
HdfsResourceLoader class, 244
HdfsTextItemWriter class, 234, 236
HdfsWritingMessageHandler class, 228
HEAD verb (HTTP), 157
Hibernate project, 37, 57
HibernateAnnotationProcessor class, 31
Hive project
  about, 195
  analyzing password file, 196
  Apache log file analysis, 202–204
  JDBC client, 201
  running Hive servers, 197
  Thrift client, 198–200
  workflow considerations, 242
Hive servers, 197
<hive-server /> element, 197
HiveClient class, 199
HiveDriver class, 201
HiveQL language, 195, 196
HiveRunner class, 201, 203
HiveServer class, 197
HiveTemplate class, 199, 202
HKEYS command, 133
Homebrew package manager, 127
href (hypertext reference) attribute, 161
hsqldb directory, 54
HSQLDB relational database, 148
HSQLDBTemplates class, 57
HTable class, 215
HTTP protocol, 157
hypermedia, 158
HyperSQL relational database, 54

I

@Id annotation, 83
identifiers, resources and, 157
IncorrectResultSizeDataAccessException class, 48
@Index annotation, 91
@Indexed annotation, 105, 107

indexes
  MongoDB support, 91
  Neo4j support, 102
  Spring Data Neo4j support, 113, 120
IndexRepository interface
  findAllByPropertyValue method, 115
  findAllByQuery method, 115
  findAllByRange method, 115
InputStream class, 201, 244
insert operations
  inserting data into MongoDB, 79
  inserting objects in JDBC, 71
IntelliJ IDEA, 9, 24, 143
Inter Type Declarations (ITDs), 141, 142
IntSumReducer class, 180
IntWritable class, 184
ITDs (Inter Type Declarations), 141, 142
ItemProcessor interface, 233, 247
ItemReader interface, 233
ItemWriter interface, 233, 247, 249
Iterable interface, 116

J

JacksonJsonRedisSerializer class, 134
Java client (HBase), 215–217
Java language
  AspectJ extension, 105, 123, 141
  MongoDB support, 80
  Neo4j support, 103
  Spring Roo tool, 141–154
Java Persistence API (JPA)
  about, 4, 37
  repositories and, 37–52
  Spring Roo example, 147–152
Java Transaction Architecture (JTA), 102
javax.persistence package, 31
javax.sql package, 57
JDBC
  about, 53
  Cypher query language and, 115
  exporting data from HDFS, 243–248
  Hive project and, 197
  insert, update, delete operations, 71–73
  query execution and, 64–71
  QueryDslJdbcTemplate class, 63–64
  sample project and setup, 54–62
JDBC client (Hive), 201
<jdbc:embedded-database> element, 54
JdbcBatchItemWriter class, 247, 249
JdbcCursorItemReader class, 235
JdbcItemReader class, 234
JdbcItemWriter class, 233
JdbcPagingItemReader class, 236
JdbcTemplate class, 196
  about, 201
  GemfireTemplate and, 269
  Querydsl and, 53, 63
JDK Timer, 191
JdkSerializationRedisSerializer class, 136
JDOAnnotationProcessor class, 31
Jedis driver, 129
JedisConnectionFactory class, 129
Jetty plug-in, 158
JmsTemplate class, 269
Job class (MapReduce), 180, 184
Job interface (Spring Batch), 232
job scheduling, 191–193
JobDetail class, 193
JobDetail interface, 193
JobLauncher interface, 232
JobRepository interface, 232
JobRunner class, 184, 191
Johnson, Rod, 3, 105
@JoinColumn annotation, 40
JPA (Java Persistence API)
  about, 4, 37
  repositories and, 37–52
  Spring Roo example, 147–152
JPA module (Querydsl)
  bootstrapping sample code, 44–46
  querying relational stores, 32
  repository usage, 37, 47–52
  sample project, 37–42
  traditional repository approach, 42–44
JPAAnnotationProcessor class, 31, 51
JPAQuery class, 32
JpaTransactionManager class, 46
JSON format, 79
@JsonIgnore annotation, 167
@JsonSerialize annotation, 166
JTA (Java Transaction Architecture), 102
JtaTransactionManager class, 115, 122
Just Spring Data Access (Konda), 53

K

key/value data (see Spring Data Redis project)
Konda, Madhusudhan, 53

L

LIKE operator, 49
LineAggregator interface, 237
List interface, 116
ListOperations interface, 130
log files
  analyzing using Hive, 202–204
  analyzing using Pig, 212–213
  loading into HDFS, 222–226
Long, Josh, 141
ls command, 178, 187
Lucene search engine library, 102, 108

M

m2eclipse plug-in, 7
@ManyToOne annotation, 41, 171
Map interface, 87, 133, 257
@MappedSuperclass annotation, 38
Mapper class, 180, 184
mapping subsystem (MongoDB)
  about, 83
  customizing conversion, 91–93
  domain model, 83–89
  index support, 91
  setting up infrastructure, 89–91
MappingContext interface, 89, 91, 93
MappingMongoConverter class, 89, 90, 93
MappingPdxSerializer class, 272
MappingProjection class, 57, 64
MapReduce API (see Hadoop MapReduce API)
MATCH identifier (Cypher), 114
Maven projects
  Appassembler plug-in, 182
  installing and executing, 6
  IntelliJ IDEA and, 9
  Jetty plug-in, 158
  m2eclipse plug-in and, 7
  maven-apt-plugin, 30, 51
  Querydsl integration, 30, 58
  querydsl-maven-plugin, 58
  Spring Roo and, 145
maven-apt-plugin, 30, 51
Mayzak, Steve, 141
media types, 157
MessageHandler interface, 212
MessageListener interface, 135
MessageListenerAdapter class, 135
messages, listening and responding to, 135–136
MessageStore interface, 229
META-INF directory, 16, 179
MethodInvokingJobDetailFactoryBean class, 193
Mongo class, 80
<mongo:mapping-converter /> element, 90, 93
MongoAnnotationProcessor class, 31, 99
MongoConverter interface, 93
MongoDB
  about, 77
  accessing from Java programs, 80
  additional information, 80
  downloading, 78
  exporting data from HDFS, 249–250
  inserting data into, 79
  mapping subsystem, 83–93
  querying with, 32
  repository example, 96–100
  setting up, 78
  setting up infrastructure, 81–82
  using shell, 79
MongoDB shell, 79
MongoDbFactory interface, 82, 94
MongoItemWriter class, 249
MongoMappingContext class, 89, 90
MongoOperations interface, 249
  about, 94
  delete method, 95
  findAll method, 95
  findOne method, 95
  geoNear method, 95
  mapReduce method, 95
  save method, 95
  updateFirst method, 95
  updateMulti method, 95
  upsert method, 95
MongoTemplate class, 82, 94–96, 249
MultiResourceItemReader class, 245

N

Neo Technology (company), 104
Neo4j graph database, 102–104, 102
  (see also Spring Data Neo4j project)
Neo4j Server
  about, 103
  usage considerations, 124

web interface, 103
<neo4j:config /> element, 112, 122
Neo4jTemplate class
about, 105, 112
delete method, 115
fetch method, 113
findAll method, 115
findOne method, 106, 113, 115
getNode method, 113
getOrCreateNode method, 107, 113
getRelationshipsBetween method, 113
load method, 113
lookup method, 107, 108
projectTo method, 113
query method, 109, 113
save method, 109, 113, 115
traverse method, 113
@NodeEntity annotation, 105, 106
Noll, Michael, 177
NonTransientDataAccessException class, 201
@NoRepositoryBean annotation, 21
NoSQL data stores
about, xvii
HDFS and, 176
MongoDB, 77–100
Neo4j graph database, 101–126
Redis key/value store, 127–137
Spring Data project and, 3–4

O

object conversion, 130–132
object mapping, 132–134
Object Query Language (OQL), 269
Object-Graph-Mapping, 105, 106–111
ObjectID class, 83
objects
deleting, 73
inserting, 71
persistent domain, 111–113
updating, 72
value, 39, 83
ObjectSizer interface, 269
@OneToMany annotation, 40
OneToManyResultSetExtractor class, 67–68
OpenJpa project, 37
OPTIONS verb (HTTP), 157
OQL (Object Query Language), 269
ORDER BY identifier (Cypher), 115
OxmSerializer class, 134

P

Page interface, 116
Pageable interface, 18, 49, 98
PageRequest class, 49, 98
PagingAndSortingRepository interface
about, 19
JPA repositories and, 50
MongoDB repositories and, 99
REST repository exporter and, 168, 170
@Parameter annotation, 116
PARAMETERS identifier (Cypher), 115
Parastatidis, Savas, 158
PartitionResolver interface, 265
PassThroughFieldExtractor class, 236
PDX serialization library (Gemfire), 255, 262, 272
Penchikala, Srini, 141
persistence layers (Spring Roo)
about, 143
setting up JPA persistence, 148
setting up MongoDB persistence, 153
persistence.xml file, 45
@PersistenceCapable annotation, 31
@PersistenceConstructor annotation, 271
@PersistenceContext annotation, 43
PersistenceException class, 269
@PersistenceUnit annotation, 45
Pig Latin language, 204, 206
Pig project
about, 195, 204
analyzing password file, 205–207
Apache log file analysis, 212–213
calling scripts inside data pipelines, 211
controlling runtime script execution, 209–211
installing, 205
running Pig servers, 207–209
workflow considerations, 243
Pig servers, 207–209
Piggybank project, 213
PigRunner class, 208
PigServer class, 204, 207–209, 210
PigStorage function, 206
PigTemplate class, 209, 210
PIG_CLASSPATH environment variable, 205
POST verb (HTTP), 157, 160
Predicate interface, 52, 57
@Profile annotation, 95
ProgramDriver class, 179, 181

Project Gutenberg, 177
Properties class, 261
properties file, 16
property expressions, 17
publish-subscribe functionality (Redis), 135–136
push-in refactoring, 143
Put class, 215
PUT verb (HTTP), 157, 160, 163

Q

QBean class, 57, 64
qualifiers (HBase), 215
Quartz job scheduler, 191, 192
@Query annotation
  about, 16
  annotated finder methods and, 116
  manually defining queries, 49
  repository support, 114
  Spring Data Neo4j support, 105
Query class, 95, 97
query execution
  extracting list of objects, 68
  OneToManyResultSetExtractor class, 67–68
  querying for list of objects, 71
  querying for single object, 65–66
  repository implementation and, 64
  RowMapper interface, 69–70
  Spring Data GemFire support, 273
query metamodel
  about, 27–30
  generating, 30–32, 51–52
query methods
  executing, 15
  pagination and sorting, 17
  property expressions, 17
  query derivation mechanism, 16–17
  query lookup strategies, 16
query types, generating, 54–57
Querydsl framework
  about, 27–30
  build system integration, 58
  database schema, 59
  domain implementation of sample project, 60–62
  executing queries, 64–71
  generating query metamodel, 30–32, 51–52
  integrating with repositories, 32–34, 51–52
  JPA module, 32, 37–52
  MongoDB module, 99
  reference documentation, 58
  SQL module, 54–57
  SQLDeleteClause class, 73
  SQLInsertClause class, 71
  SQLUpdateClause class, 72
querydsl-maven-plugin, 58
QuerydslAnnotationProcessor class, 31
QueryDslJdbcTemplate class
  about, 63–64
  delete method, 73
  executing queries, 64, 66, 69
  insert method, 71
  insertWithKey method, 71
  update method, 72
QueryDslPredicateExecutor interface, 33, 52, 100
@QueryEmbeddable annotation, 30, 31
@QueryEntity annotation, 28, 30, 31
QueryService interface, 269

R

rapid application development
  REST repository exporter, 157–171
  Spring Roo project, 141–154
Redis key/value store
  about, 127
  atomic counters, 134
  cache abstraction, 136
  connecting to, 129
  object conversion, 130–132
  object mapping, 132–134
  publish/subscribe functionality, 135–136
  reference documentation, 129
  setting up, 127–128
  using shell, 128
Redis shell, 128
RedisAtomicLong class, 134
RedisCacheManager class, 136
RedisOperations interface, 132
RedisSerializer interface, 131, 132, 135
RedisTemplate class
  about, 130
  opsForHash method, 133
  RedisCacheManager class and, 136
Reducer class, 180, 184
refactoring, push-in, 143

Region interface, 257, 269
regions
  caches and, 257
  configuring, 263–265
  Gemfire-supported, 257
rel (relation type) attribute, 161
@RelatedTo annotation, 105, 107, 110, 123
@RelatedToVia annotation, 110, 123
relational databases
  deleting objects, 73
  HSQLDB, 148
  HyperSQL, 54
  inserting data, 71
  JDBC programming, 53–73
  JPA repositories, 37–52, 147–152
  loading data to HDFS, 234–237
  problem of scale, xv
  problems with data stores, xvi
  updating objects, 72
@RelationshipEntity annotation, 105, 109, 110
repositories, 18
  (see also REST repository exporter)
  basic graph operations, 115
  creating, 272
  defining, 18–19, 150, 154
  defining query methods, 16–18
  derived and annotated finder methods, 116–119
  fine-tuning interfaces, 20
  IDE integration, 22–24
  integrating Querydsl with, 32–34, 51–52
  integrating with Spring Data Neo4j, 113–119
  JPA implementation, 37–52, 147–152
  manually implementing methods, 21–22, 34
  MongoDB implementation, 96–100, 152–154
  query execution and, 64
  quick start, 13–15
  Spring Data GemFire and, 271–273
  Spring Roo examples, 147–154
  traditional approach, 42–44
<repositories /> element, 14
@Repository annotation
  data access and, 269
  enabling exception translations, 43
  Hive Thrift client and, 198
  making components discoverable, 45, 65, 95
Repository interface
  about, 14, 19
  executing predicates, 33
@RepositoryDefinition annotation, 20
Representational State Transfer (REST), 157
representations, defined, 157
resources and identifiers, 157
REST (Representational State Transfer), 157
REST repository exporter
  about, 157
  sample project, 158–171
REST web services, 158
@RestResource annotation, 161, 164
@ResultColumn annotation, 117
ResultConverter interface, 117
ResultScanner interface, 216
ResultSet interface, 201, 235
ResultSetExtractor interface, 64, 68
ResultsExtractor interface, 216
RETURN identifier (Cypher), 115
RFC 4287, 161
Riak database, 127
Rimple, Ken, 141
rmr command, 182
Robinson, Ian, 158
Roo Shell
  about, 146
  creating directories, 153
Roo shell
  creating directories, 148
roo-spring-data-jpa directory, 148
roo-spring-data-mongo directory, 153
@RooJavaBean annotation, 141, 142, 149
@RooJpaEntity annotation, 149
@RooToString annotation, 149
RowMapper interface, 64
  HBase and, 216
  implementation examples, 69–70
  MappingProject class and, 57

S

sample project
  bootstrapping sample code, 44–46
  JDBC programming, 54–62
  JPA repositories, 37–42
  mapping subsystem, 83–93
  REST expository exporter, 158–171

Scan class, 215
scheduling jobs, 191–193
<script /> element, 188, 189
sed command, 177, 204
Serializable interface, 137, 272
@Service annotation, 45
ServletContext interface, 160
SET commands (Redis), 129
Set interface, 116
SetOperations interface, 130
show dbs command, 79
SimpleDriverDataSource class, 201
SimpleJdbcTestUtils class, 202
SimpleJpaRepository class, 47, 48, 50
SimpleMongoDbFactory class, 82
SimpleMongoRepository class, 97
SKIP LIMIT identifier (Cypher), 115
SkipListenerSupport class, 245
Sort class, 18
SpEL (Spring Expression Language), 189, 230, 241
Splunk, 250–252
  product website, 250
  sample application, 251
Spring Batch project
  about, 176, 219, 232–234
  additional information, 232
  Hadoop support, 238–240
  Hive and, 242
  Pig and, 205, 243
  processing and loading data from databases, 234–237
  wordcount application and, 240–242
Spring Data GemFire
  about, 255
  cache client configuration, 265
  cache configuration, 258–262
  cache server configuration, 267
  configuration considerations, 258
  continuous query support, 273
  data access with, 269
  disk store configuration, 268
  region configuration, 263–265
  repository usage, 271–273
  WAN configuration, 267–268
Spring Data JDBC Extensions sub-project, 67
Spring Data JDBC Extensions subproject, 53
Spring Data Neo4j project
  about, 105
  additional information, 126
  advanced graph use cases, 119–122
  advanced mapping mode, 123–124
  combining graph and repository power, 113–119
  entity lifecycle, 122
  fetch strategies, 122
  Java-REST-binding, 105
  modeling domain as graph, 106–111
  Neo4j server and, 124
  persistent domain objects, 111–113
  transactions and, 122
Spring Data project
  about, 3
  domain model, 6
  general themes, 5
  NoSQL data stores and, 3–4
  Querydsl integration, 27–34
  repository abstraction, 13–24, 37–52
  sample code, 6–11
Spring Data Redis project
  atomic counters, 134
  caching functionality, 136
  connecting to Redis, 129
  object conversion, 130–132
  object mapping, 132–134
  publish/subscribe functionality, 135–136
Spring Data REST repository exporter project (see REST repository exporter)
Spring Expression Language (SpEL), 189, 230, 241
Spring for Apache Hadoop
  about, 176
  combining HDFS scripting and job submission, 190
  configuring and running Hadoop jobs, 183–187
  embedding PigServer, 205
  goal for, 219
  Hive and, 196
  job scheduling, 191–193
  Pig and, 207
  scripting features, 175
  scripting HDFS on the JVM, 187–190
  Spring Integration and, 222
Spring Integration project
  about, 176, 219, 220–222
  additional information, 222
  calling Pig scripts inside data pipelines, 211

copying log files, 222–226
event forwarding and, 229
event streams and, 226–229
key building blocks, 220
management functionality, 221, 230–231
Pig and, 205
processing pipeline example, 221
Spring MVC Controller, 185
Spring Roo in Action (Rimple and Penchikala), 141
Spring Roo project
about, 141–143
additional information, 141
downloading, 143
JPA repository example, 147–152
MongoDB repository example, 152–154
persistence layers, 143
quick start, 143
Spring Tool Suite (see STS)
spring-jdbc module, 53, 54
SpringRestGraphDatabase class, 125
SQL module (Querydsl), 54–57
SqlDeleteCallback interface, 73
SQLDeleteClause class, 73
SQLInsertClause class, 71
SqlInsertWithKeyCallback interface, 72
SQLQueryImpl class, 57
SqlUpdateCallback interface, 72
SQLUpdateClause class, 72
START identifier (Cypher), 114
Step interface, 232, 238
String class
accessing data via, 27
converting binary values, 83
Map interface and, 87
Spring Data Redis and, 130
StringRedisTemplate class, 130
STS (Spring Tool Suite)
creating Spring Batch jobs, 238
m2eclipse plug-in and, 7
MongoDB repository example, 153
repository abstraction integration, 22–24
Spring Roo JPA repository example, 147
Spring Roo support, 145
syslog file, 226

T

TableCallback interface, 216
TaskExecutor interface, 212
Tasklet interface, 238, 243
TaskScheduler interface, 191–192
10gen (company), 80
Text class, 184
ThreadPoolTaskScheduler class, 191
Thrift client (Hive), 198–200
TokenizerMapper class, 180
TRACE verb (HTTP), 157
transaction managers, 122
@Transactional annotation
annotating methods, 45
CRUD operations and, 50
data access and, 269
defining transactional scopes, 122
setting flags, 43, 65
wrapping method calls, 65
TransactionLister interface, 262
transactions
activating, 45
CRUD operations and, 50
read-only methods and, 65
Spring Data Neo4j and, 122
verifying execution of, 43
wrapping method calls, 65
TransactionWriter interface, 262
@Transient annotation, 271
TransientDataAccessException class, 201
TraversalRepository interface, 116
traversals
graph databases, 101
Neo4j support, 102
Spring Data Neo4j support, 113
Trigger interface, 191

U

UPDATE identifier (Cypher), 115
updating objects, 72
util namespace, 262

V

value objects, 39, 83
ValueOperations interface, 130, 132
vFabric Gemfire website, 257

W

WAN communications, 267
wc command, 177
web pages, creating, 150, 154

WebApplicationInitializer interface, 159
Webber, Jim, 158
webhdfs scheme, 183, 184
wget command, 177
WHERE identifier (Cypher), 114
wordcount application (Hadoop)
  about, 175
  introductory example, 177–183
  Spring Batch and, 240–242
  Spring for Apache Hadoop example, 183–187
WordCount class
  introductory example, 180–183
  Spring for Apache Hadoop example, 183
workflows
  about, 238
  executing Hive scripts, 242
  executing Pig scripts, 243
  Spring Batch support, 238–240
  wordcount application and, 240–242
WorkManager interface, 192
WriteConcern class, 82

X

XA transaction managers, 122
XML namespace elements
  activating JPA repositories through, 47
  activating repository mechanisms, 96
  base-package attribute, 14, 93
  db-factory-ref attribute, 90
  repository support, 22
  setting up MongoDB infrastructure, 81–82
  Spring Data Neo4j project, 105

Z

ZSetOperations interface, 130

About the Authors

Dr. Mark Pollack worked on big data solutions in high-energy physics at Brookhaven National Laboratory and then moved to the financial services industry as a technical lead or architect for front-office trading systems. Always interested in best practices and improving the software development process, Mark has been a core Spring (Java) developer since 2003 and founded its Microsoft counterpart, Spring.NET, in 2004. Mark now leads the Spring Data project that aims to simplify application development with new data technologies around big data and NoSQL databases.

Oliver Gierke is an engineer at SpringSource, a division of VMware, and project lead of the Spring Data JPA, MongoDB, and core module. He has been involved in developing enterprise applications and open source projects for over six years. His working focus is centered on software architecture, Spring, and persistence technologies. He speaks regularly at German and international conferences and is the author of several technology articles.

Thomas Risberg is currently a member of the Spring Data team, focusing on the MongoDB and JDBC Extensions projects. He is also a committer on the Spring Framework project, primarily contributing to enhancements of the JDBC framework portion. Thomas works on the VMware's Cloud Foundry team, developing integration for the various frameworks and languages supported by the Cloud Foundry project. He is coauthor of *Professional Java Development with the Spring Framework*, together with Rod Johnson, Juergen Hoeller, Alef Arendsen, and Colin Sampaleanu, published by Wiley in 2005.

Jon Brisbin is a member of the SpringSource Spring Data team and focuses on providing developers with useful libraries to facilitate next-generation data manipulation. He's helped bring elements of the Grails GORM object mapper to Java-based MongoDB applications, and has provided key integration components between the Riak datastore and the RabbitMQ message broker. In addition, he blogs and speaks on evented application models, and is working diligently to bridge the gap between the bleeding-edge nonblocking and traditional JVM-based applications.

Michael Hunger has been passionate about software development for a long time. He is particularly interested in the people who develop software, software craftsmanship, programming languages, and improving code. For the last two years, he has been working with Neo Technology on the Neo4j graph database. As the project lead of Spring Data Neo4j, he helped develop the idea for a convenient and complete solution for object graph mapping. He also takes care of Neo4j cloud-hosting efforts. As a developer, Michael loves working with many aspects of programming languages, learning new things every day, participating in exciting and ambitious open source projects, and contributing to different programming-related books. Michael is also an active editor and interviewer at InfoQ.

Colophon

The animal on the cover of *Spring Data* is the giant squirrel (genus *Ratufa*), which is the largest squirrel in the world. These squirrels are found throughout tropical Asiatic forests and have a conspicuous two-toned color scheme with a distinctive white spot between the ears. Adult head and body length varies around 14 inches and the tail length is approximately 2 feet. Their ears are round and they have pronounced paws used for gripping.

A healthy adult weighs in at around four and a half pounds. With their tan, rust, brown, or beige coloring, they are possibly the most colorful of the 280 squirrel species. They are herbivorous, surviving on flowers, fruits, eggs, insects, and even bark.

The giant squirrel is an upper-canopy dwelling species, which rarely leaves the trees, and requires high branches for the construction of nests. It travels from tree to tree with jumps of up to 20 feet. When in danger, the giant squirrel often freezes or flattens itself against the tree trunk, instead of fleeing. Its main predators are birds of prey and leopards. The giant squirrel is mostly active in the early hours of the morning and in the evening, resting in the midday. It is a shy, wary animal and not easy to discover.

The cover image is from *Shaw's Zoology*. The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSansMonoCondensed.